はじめて読むC言語

蒲地 輝尚 著

アスキー出版局

Photographed by Akio Kambayashi
Designed by Keiko Go

はじめて読む
C言語

蒲地輝尚　著

アスキー出版局

商　標

・MS-DOS、Microsoft C Professinal Development System、Microsoft Quick C Compiler は、米 Microsoft 社の商標です。
・UNIX は、米 AT&T のベル研究所が開発し、AT&T がライセンスしています。
・TURBO C は、米 BORLAND 社の商標です。
そのほか、CPU 名、システム名などは一般に各開発メーカーの商標です。なお、本文中では、TM、Ⓡマークは明記していません。

カバーデザイン：郷　啓子　カバー写真：神林 章夫　イラスト：三善 和彦
図版作成：保坂 庸介／長谷川 弘／緒方 由希子（山岡デザイン事務所）

本書を読む前に

本書は、コンピュータのプログラムを記述するためのプログラミング言語の１つである、C言語の解説書です。本書の解説は、C言語を利用できるすべてのコンピュータを対象としています。

本書の内容を学習するために必要な知識は、ごく一般的なコンピュータの操作方法だけで十分です。ワードプロセッサなどのアプリケーションを利用した経験があればなおよいでしょう。

プログラミングの経験はいっさい必要ありません。他の言語でのプログラミング経験があれば、本書の解説はより容易に理解できるでしょう。しかし、まったく経験のない方でも理解できるように、初歩から解説を行っています。本書のプログラムを実際に実行して試してみるためには、C言語を利用できるコンピュータが必要です。手近のコンピュータでC言語を利用できない方は、別途入手してください。パーソナルコンピュータ用には、C言語パッケージが多数販売されています。

本書の例題プログラムは、C言語を利用できるほとんどのコンピュータでそのまま実行することができます。C言語に関する規格が制定される以前から利用されているC言語のパッケージでは、書式に若干の違いがありますが、必要な時点で解説を行っているので、適切な変更を加えることで本書のプログラムを実行して試してみることができます。

はじめに

パーソナルコンピュータの登場によって、コンピュータはたいへん身近な存在になりました。その利用分野は、ワードプロセッサをはじめ表計算やデータベースなど、多岐にわたっています。さらに、実用的な面ばかりでなく、知的好奇心をかきたて創作意欲を湧き起こす媒体としての役割も見逃せません。プラモデル作りや日曜大工にも通じる楽しさを見い出せることが、コンピュータの魅力の１つです。筆者も、こうした魅力に惹かれてコンピュータを楽しんでいます。

プログラミング言語によって、私たちは新しいソフトウェアを創り出す発明家になることができます。実用的なアプリケーションを作成することも可能ですし、ソフトウェア工学への理解を深めて知的好奇心を満足させることにもつながります。読者のみなさんは、ぜひこうした楽しみを存分に味わってほしいと思います。

プログラミング言語の中でも、Ｃ言語は最も広く普及し、多くの分野で使われている言語です。当初、Ｃ言語はプロやマニアが使う玄人向けのプログラミング言語として位置づけられていました。ところが現在は、初心者も含めた標準的なプログラミング言語として利用されています。

ここでよく問題にされるのが、Ｃ言語は初心者向きでないという意見があることです。それどころか、他のプログラミング言語を習得している人であっても、新たにＣ言語を習得するのは難しいとさえいわれています。

その理由の多くは、コンピュータの仕組みに密着した、マシン語と呼ばれる言語を知らなければ、Ｃ言語をマスターすることはできないというものです。Ｃ言語には、確かにこうした側面があり、コンピュータの仕組みに関するある程度専門的な知識が必要なのは、まさにその通りです。しかし、これはマシン語を知っていなければならないということとはちょっと違います。マシン語を習得しておくのが望ましいことは確かですが、必ずしも詳細な知識が必要なわけではありません。

マシン語を知っている人は、コンピュータの仕組みについてのイメージを頭の中に持っています。そのイメージがあるからこそ、Ｃ言語を容易に理解

し、使いこなすことができます。重要なのは、マシン語そのものよりも、そうしたイメージを持つことです。C言語のプログラムから、コンピュータの仕組みに対応したイメージ図を思い浮かべられることが必要なのです。

本書では、みなさんの頭の中にこうしたイメージを形作ってもらうことを大きな目標としています。図版を多用して丁寧に解説しているので、本書を読み進むうちにコンピュータの真の姿が自然と見えてくるでしょう。プログラムを眺めて、それがコンピュータの中でどういう姿をしているかを思い浮かべられるようになれば、もはやC言語をマスターしたといっても過言ではありません。

本書では、C言語のすべてを解説しているわけではありませんが、現実のプログラミングで使われる主要な機能について重点的に解説しています。本書の内容をマスターすれば、通常のプログラムのほとんどを読みこなせるようになると思ってよいでしょう。

本書は、コンピュータの世界における事実上の共通語である、C言語を習得する上で大きな力になれると期待しています。

1991 年 3 月

蒲地輝尚

目　次

第1章

C 言語の基礎知識

“

これからみなさんは、C言語を征服する登山をはじめるところです。この山は一部険しいところがあり、遭難してしまう人も少なくありません。しかし、この山の頂上からの眺めはとてもすばらしいものです。本書は、頂上への道が見えるところまでみなさんを導くガイド役としての役割をきっと果たせるでしょう。

登山をはじめる前に、地図を広げて山の地形全体をおおまかに把握しておくことにします。本章で解説するのは、C言語が広く普及している背景と、C言語プログラムを実行させるまでの作業手順です。

C言語のすばらしさは、実際にプログラムを作成してみなければ、なかなかつかむことはできませんが、その背景にある秘密に触れておくことはC言語を学習する上で必ず役に立つでしょう。

”

1.1

プログラミング言語としてのC

コンピュータとプログラム

コンピュータは**ハードウェア**と**ソフトウェア**を組み合わせることによって動作する装置です。ハードウェアとは、コンピュータの装置そのものを指し、電子部品を組み合わせた機械としてのコンピュータを意味します。ソフトウェアはハードウェアに読み込ませることによってコンピュータの動作をコントロールする、**プログラム**のことです。

パーソナルコンピュータを利用している方なら、ソフトウェアパッケージを買ってきて読み込ませることで、コンピュータをさまざまな目的に利用できることをよく理解しているでしょう。そして、もちろん、買ってきたソフトウェアばかりでなく、自分でソフトウェアを作成して動作させることもできるのです。

ソフトウェアを作成するということは、プログラムを作成することです。文章を書くことになぞらえて「プログラムを書く」といったり、部品を組み立てることになぞらえて「プログラムを組む」ということもあります。

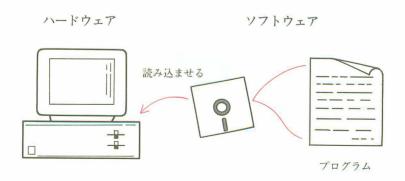



図1-1　コンピュータとソフトウェア

プログラミング言語とは

コンピュータは電子部品を組み合わせた機械（マシン）です。したがって、コンピュータに直接指示できるのは、**マシン語**と呼ばれる電子的な命令だけです。コンピュータのソフトウェアの実体は、マシン語命令の集まりなのです。

しかし、そういったマシン語のレベルでコンピュータに与える指示をプログラムするのは想像するだけでもたいへんなことです。そこで、図1-2のように、数式のような人間にわかりやすいかたちでプログラムを作成し、それを一連のマシン語命令に変換してから、実行する方式が考え出されました。

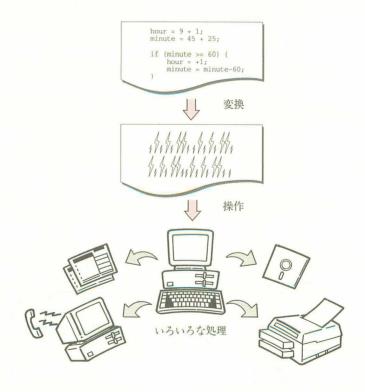



図1-2　プログラミング言語とマシン語

プログラムを書くための形式は、目的に応じていくつも考え出されました。それが**プログラミング言語**であり、C言語もそうしたプログラミング言語の1つです。コンピュータはマシン語レベルの指示しか受け取りませんが、私たちは、あたかもプログラミング言語で書かれたプログラムでコンピュータを操作している、と考えるのです。

C言語の特徴

プログラミング言語は、処理の目的に応じてプログラムの形式に工夫が凝らされています。図1-3のように、プログラミング言語ごとにプログラムの書き方や情報の表現方法、プログラム実行の進め方などに特徴があり、それぞれ得意な分野と苦手な分野があります。

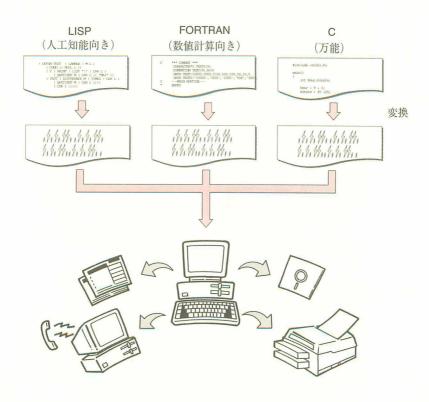



図1-3　各プログラミング言語の特徴

たとえば、FORTRAN（フォートラン）という言語は、物理学や工学の世界で科学技術計算を主な用途として使われています。COBOL（コボル）は、事務計算を中心に広く使われています。LISP（リスプ）は、さまざまな概念の構造を表現しやすい独特の表記を採用し、言語処理や人工知能などの研究に応用されています。

さて、こうしたプログラミング言語の中で、C言語の特徴は、以下のようなものが挙げられるでしょう。

●マシンレベルに近い記述能力

多くのプログラミング言語は、コンピュータの機械的な仕組みをあまり意識せずに、概念的にコンピュータを操作できるように設計されています。C言語も基本的には同じ考え方で設計されているのですが、コンピュータの機械的な仕組みを直接利用する機能も持っています。概念的な処理のイメージをうまく保ちながら、ぎりぎりまで機械的なレベルの仕組みに近づいた指示を書くことができるのです。

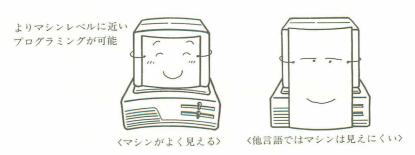



図1-4a　マシンレベルに近い処理能力

●高速な実行速度

コンピュータに最高速で処理を実行させるには、マシン語レベルの指示を直接コンピュータに与えることが一番です。プログラミング言語で書いたプログラムは、マシン語レベルの指示に変換されてから実行されますが、かならずしも最適な指示に変換されるとは限りません。これに対してC言語は、効率よくマシン語レベルの指示に変換できるように設計されています。なお、言語によっては、マシン語中にプログラムミスのチェックなどを行う指示が

自動的に挿入されるものもありますが、C言語の場合は、このようなチェックを行いません。

こうしたことに加え、マシンレベルに近い記述能力を最大限に利用すると、場合によっては、直接マシン語レベルの指示を使ってプログラムを書くのと変わらないほどの高速実行が可能になります。

図 1-4b　高速な実行速度

●シンプルな言語仕様

次節で全体像を紹介しますが、C言語の言語仕様は非常にシンプルです。他の多くの言語では、画面表示やキーボード入力などの機能を、命令文として言語の中に組み込んでいます。これに対しC言語では、こうした入出力機能を言語に組み込まず、プログラム部品として別途提供する形にしています。このため、文法は非常にすっきりしており、入出力機能を別のプログラム部品と取り替えることも簡単です。

●モジュール化に対応

モジュール化というのは、プログラムを部品として作成し、プログラム部品を組み合わせることによって大きなプログラムを作っていく方法のことです。C言語は、プログラムのモジュール化に対応しているので、あるプログラムの一部を別のプログラムで再利用したり、大規模な処理を小さな処理の組み合わせとして構築することができます。

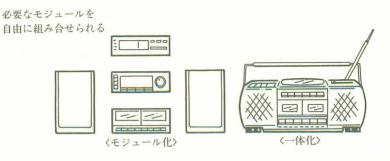



図 1-4c　モジュール型

C言語の全体像

ひとくちにプログラミング言語といっても、図1-5のように**文法**と**処理系**の2つの要素に分けることができます。

「文法」とはプログラムを書く形式を規定する決まりのことです。「処理系」とは、プログラミング言語で書かれたプログラムをマシン語に変換するソフトウェアや、関連するソフトウェアなどのことです。C言語の文法を知っていれば、C言語のプログラムを書くことはできますが、処理系がなければプログラムを実行させることはできません。

文法は一般に言語仕様として決められており、どの処理系を使っても同じ形式でプログラムを書くことができます。しかし、処理系の操作方法は処理

図 1-5　C言語の全体像（1）

系ごとに異なります。また、処理系によって同じ実行結果が得られても、作成されるマシン語プログラムの中味は異なります。

本書では主にC言語の文法を解説しますが、プログラムを実行させるまでの手順もC言語の重要な要素ですから、各処理系に共通な処理の概要についても解説します。

C言語の文法は、図1-6のように他の言語に比べて非常にシンプルです。それは、他の言語では文法の一部として規定されている部分が、C言語では処理系の一部として提供される仕組みになっているからです。

C言語の文法は、コンピュータを操作するためのエッセンスといってもよいでしょう。シンプルでかつ強力な機能を持つからこそ、あらゆる分野に応用でき、さらにコンピュータの仕組みを学ぶためにも最適なのです。

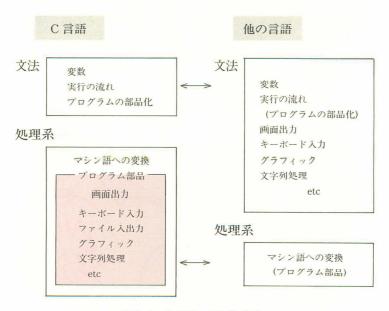
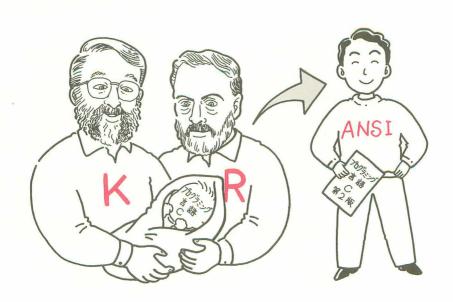



図1-6　C言語の全体像（2）

C言語の歴史

C言語には、大きく分けて2つの種類あります。ひとつは**K&R準拠のC言語**であり、もうひとつは**ANSI準拠のC言語**です。両者の関係を理解するために、C言語の歴史を解説しましょう。

C言語は、アメリカのAT&Tベル研究所で、UNIX（ユニックス）というオペレーティングシステムのために開発されたプログラミング言語です。最初のUNIXが開発された1970年頃、このようなオペレーティングシステムの記述に適したプログラミング言語はありませんでした。そこで、Dennis M. Ritchie（デニス・リッチー）は、オペレーティングシステムの記述に適したC言語を自ら開発したのです。

C言語の名前の由来には、おもしろい逸話があります。1970年にUNIX開発者の主要メンバーのKen Thompson（ケン・トンプソン）は、UNIXのために、BCPLという言語を元にB言語を開発しました。その後、開発されたプログラミング言語は、Bの次であることからCと名付けられたのです。

やがて、C言語の優れた点が認められ、一般に普及するようになりました。パーソナルコンピュータや、ワークステーションをはじめ多くのコンピュータ用にC言語処理系が開発されましたが、言語仕様をまとめた唯一の文献として参照されたのが、Brian W.Kernighan（ブライアン・カーニハン）とC言語の開発者であるリッチーの書いた解説書「プログラミング言語C」*1だったのです。この本は、著者の頭文字をとって「K&R」と呼ばれています。K&Rは、長い間C言語の規格書としての役割を果たしていたことから、この本の仕様に基づいた処理系をK&R準拠のC言語と呼びます。

K&Rの出版以後も、C言語は改良され、拡張されてきました。また、K&Rには細かいところであいまいな部分があり、言語仕様がはっきりしない点がありました。C言語がUNIXを離れて多方面で使われるようになると、新しい拡張機能を取り入れた言語仕様の規格化を望む声が高まり、検討が続けられました。そして、1989年、ANSI（American National Standards Instisute：米国国内規格協会）によって、標準規格がまとめられたのです。この規格に準拠したC言語をANSI-Cと呼びます。

ANSI-Cは、K&Rの言語仕様を多少拡張したものですが、基本的には大きな違いはありません。また、K&Rとの互換性を保っているので、K&R仕様で書かれたプログラムをそのまま実行することもできます。本書では、ANSI-Cの言語仕様に準拠して解説を行いますが、両者に差がある部分については、K&R仕様も適宜解説します。例題プログラムも、ごく一部の変更でK&R準拠のC言語でも実習することができます。

パーソナルコンピュータ用のC言語処理系も、現在、ほとんどがANSI-C準拠となっています。ただし、UNIXワークステーション用のC言語では、ANSI対応でないK&R準拠のものも少なくありません。

「K&R」も、ANSI規格に基づいた第2版が出版されています。「K&R」はC言語の解説書であると同時に、C言語の開発者である著者たちのプログラミングに対する哲学が語られており、たいへん勉強になる本です。ただし、はじめての方が読むには多少難しいので、プログラミングの経験を積んでから読んでみるとよいでしょう。

＊1　石田晴久訳、共立出版刊。原書名は、「The C Programming Language」。

C++の登場

1980年代になると、「オブジェクト指向」という考え方がクローズアップされるようになりました。オブジェクト指向を一言でいうと、情報に処理を加えるという考え方ではなく、情報自身が自分で行動するかのようにプログラムを組み立てることです。

Bjarne Stroustrup（ビジャン・ストラストラップ）は、C言語にオブジェクト指向の考え方を取り入れた「C++言語」を開発しました。「++」はC言語の演算子の1つで、『増やす』という意味があります。そこで、C言語よりも1歩進んだ言語という意味がこの名前に込められています。

このような話をすると、C言語はすでに時代遅れで、C++言語を勉強しなければならないのではないか、と思う人もいるかもしれませんが、あわてる必要はありません。C++言語は、C言語と互換性を保ちながら拡張された言語で、オブジェクト指向プログラミングに必要な機能を追加したものなのです。したがって、C++言語を学習するためには、C言語の知識は欠かせません。まず、C言語から習得しなければならないのです。また、コンピュータの仕組みを知るという意味からは、C言語の習得が最も適した方法であることに、今後も変わりはないでしょう。

C言語習得の心得

C言語を習得するには、まず第一にコンピュータの仕組みについて理解することが重要です。本書でも、コンピュータの仕組みを解説しながら、C言語の文法を解説していきます。プログラムが実行される裏にあるコンピュータの仕組みを意識しながら、理解を深めてください。

C言語を使う上で、いくつか注意することがあります。まず、C言語は、プログラムミスのチェックを、プログラム実行時に行わないので、ちょっとしたミスから思わぬ結果になることがあります。他のプログラミング言語なら、エラーメッセージを表示してプログラムの実行が停止するような場合でも、C言語では、でたらめに実行を進めてしまったり（これを**プログラムの暴走**といいます）、永遠に同じところを繰り返す無限ループになってプログラ

ムの実行を停止できなくなることがあります。こうなってしまったら、パーソナルコンピュータの場合は、リセットボタンを押すしかありません。

さらに、C言語はコンピュータの仕組みと密接な関係があるだけに、ときには危険な結果も引き起こします。パーソナルコンピュータには、すでにあるデータを不必要に書き換えてしまうことを防ぐメモリ保護機能がありませんから、プログラムミスからシステム領域を破壊してしまう可能性もあります。

このように、プログラムミスがリセットにつながる可能性は、他の言語に比べて格段に高いでしょう。

とはいえ、コンピュータ本体を破損したり、ディスクの内容を破損してしまうような事故は滅多に起こるものではありませんから、怖がる必要はありません。

1.2

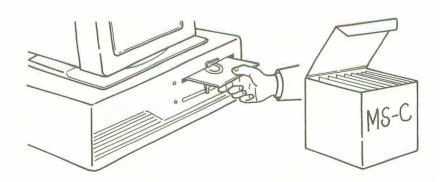
C言語プログラムを実行させるまで

C言語処理系のインストール

ひとくちにC言語と呼ばれていますが、マシン語レベルへの変換プログラムをはじめとして、さまざまなツールやファイルから構成されています。このようなC言語のシステム全体のことを、**C言語処理系**と呼びます。

本書の例題プログラムを実際に試すには、C言語処理系を利用できるコンピュータが必要ですが、会社や学校のコンピュータでC言語処理系を利用できる場合は、それを利用するとよいでしょう。本書の例題プログラムはどれも短いものばかりなので、比較的簡単に試してみることができます。

パーソナルコンピュータで実習する場合は、C言語処理系のソフトウェアパッケージを購入して、インストールしなければなりません。インストールの方法は処理系ごとに異なりますから、各処理系のマニュアルを参照してください。

また、購入したC言語処理系にテキストエディタが含まれていない場合は、テキストエディタも用意してください。テキストエディタを持っていない人は、別途購入するかワープロソフトで代用することになります。

プログラム実行までの手順

C言語プログラムを実行させるまでの、おおまかな作業手順を示したのが、図1-7です。実際の作業内容は処理系によって異なりますから、マニュアル等を参照してください。

主なC言語処理系について、Appendix 1に操作例を載せておきますので、参考にしてください。

C言語で書いたプログラムを**ソースプログラム**と呼び、そのソースプログラムをマシン語レベルに変換したものを**オブジェクトプログラム**、または**実行可能プログラム**と呼びます。

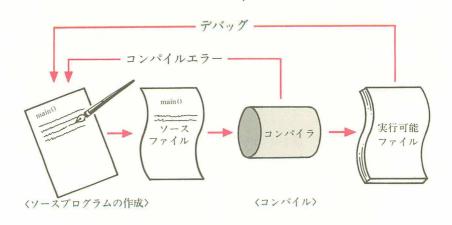



図1-7　プログラム実行までの手順

ソースプログラムの作成

ソースプログラムは、テキストエディタを使って作成します。ソースプログラムの作成は、ワープロソフトで文章を書くようなものです。プログラムの入力はもちろん、プログラムミスの修正や改造なども、テキストエディタを使います。

ソースプログラムのファイル名には、図1-8のように末尾に「.c」を付けます。

ワープロソフトを使ってプログラムを入力することもできますが、なるべくならテキストエディタを利用することをお薦めします。なぜなら、テキス

トエディタは、ソースプログラムの入力に適した機能を持っているからです。

ワープロソフトを使う場合は、全角文字を使わずに半角文字で入力することに気をつけてください。とくに、空白を全角で入力してしまうと、なかなか気がつかないので注意が必要です。

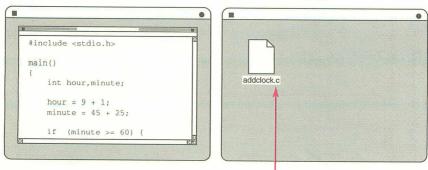
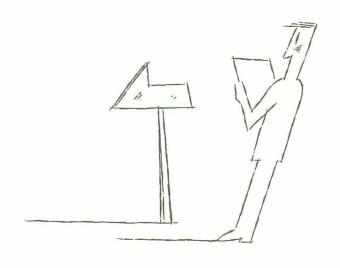



図 1-8　ソースプログラムの作成

コンパイル

次に、ソースファイルを実行可能ファイルに変換する作業をします。このことを、**コンパイル**といい、コンパイルを行うコマンドのことを**コンパイラ**と呼びます。

コンパイル作業を実行する様子を、図1-9に示します。コンパイルの方法は処理系ごとに異なりますから、マニュアルを参照してください。

コンパイルが済むと、図のように実行可能ファイルが作成されます。このファイルは、システム標準のコマンドと同じように、コマンドとして実行することができます。

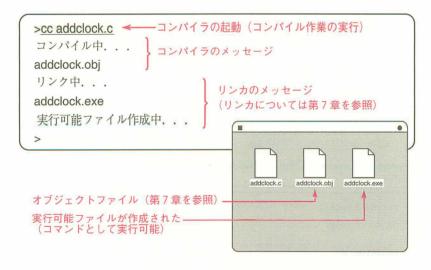



図1-9　コンパイル

コンパイルエラー

ソースプログラムに入力ミスなどがあると、コンパイル中にコンパイルエラーが発生します。図1-10はソースプログラム中で「；（セミコロン）」記号を入力し忘れたために、コンパイルエラーが発生した様子を示しています。

このようなコンパイルエラーが発生しても、あわてることはありません。エラーが起こると「エラーメッセージ」が表示され、エラーの種類やエラーを起こした行の行番号がわかるので、それを参考にエラー箇所を探します。

本書に収めた例題のプログラムであれば、入力したソースファイルとプログラムリストをよく見比べて、入力ミスをチェックしてください（エラーとして表示された行よりも前の行に、誤りがあることも多いので注意）。プログラムミスを見つけたら、テキストエディタで修正して再度コンパイルします。こうして、コンパイルエラーが起こらなくなるまで、修正とコンパイルを繰り返します。

プログラム作成中は、かならずコンパイルエラーが出ることを覚悟しなければなりません。どんなに熟練したプログラマーでも、タイプミスやカッコの対応の間違いから、コンパイルエラーを起こすものです。

```
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1;
    minute = 45 + 25;

    if (minute >= 60) {
        hour = hour + 1;
        minute = minute - 60
    }
    printf("%d:%d\n",hour,minute);
}
```

行末の「；」（セミコロン）が抜けている

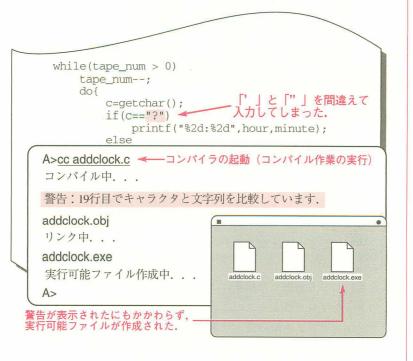
```
>cc addclock.c
コンパイル中. . .

12行目でエラーが発生しました.
コンパイルを中断します.
>
```

cc addclock.c：コンパイラの起動（コンパイル作業の実行）

コンパイラのエラーメッセージ

図1-10　コンパイルエラー

警告メッセージ

処理系によっては、コンパイルエラー以外に「警告（Warning）」メッセージが出る機能を持っているものがあります。

上の図は、警告メッセージが出力された例です。このプログラム例では、「'?'」と書くべきところを「"?"」と書いてしまったので、実行しても意図した結果にはなりません。

C言語の文法は、非常に自由度が高く便利ですが、それだけにプログラムミスを起こすことも少なくありません。警告メッセージは、エラーではないものの、プログラムミスの可能性のある部分を親切に指摘してくれているのです。

プログラムの実行

コンパイルによって作成された実行可能ファイルは、コンピュータがそのまま実行できる形式になっています。これは、あたかもシステムに用意されているコマンドであるかのように実行することが可能です（図1-11参照）。プログラムを実行させる方法は、処理系やオペレーティングシステムによって異なりますから、マニュアルを参照してください。

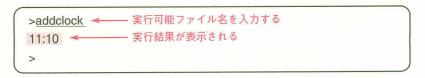



図1-11　プログラムの実行

デバッグ

プログラムを実行してみると、意図したとおりに動作しないことがあります。なぜなら、プログラムを書く時に、タイプミスや勘違いから誤った処理手順を書いてしまうことがあるからです。

例えば、次の図1-12は＋と－を入れ換えてしまったものです。このプログラムを実行しても意図した結果にはなりません。

このようなプログラムミスのことを**バグ**(bug)と呼びます。バグとは虫のことで、プログラムを喰い荒らす害虫というイメージです。

プログラムにバグがあることがわかったら、原因をつきとめて修正しなければなりません。この作業を**デバッグ**と呼びます。バグを取り除く作業です。

本書のプログラムを実際に入力して実習する場合には、入力したソースプログラムと本書のソースプログラムリストとを見比べて、バグを探してください。

バグを発見したら、ソースプログラムを修正し、コンパイル以降の作業を再度行います。プログラムがうまく実行されるようになったら、晴れてプログラムの完成というわけです。

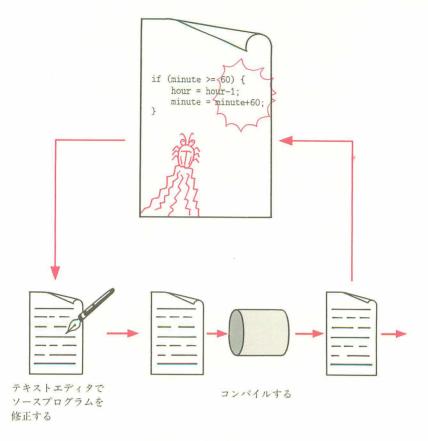
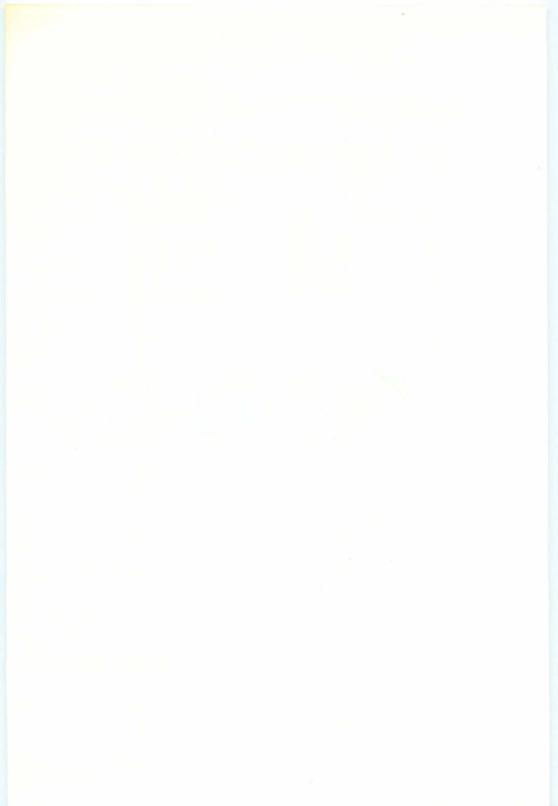



図 1-12 バグとデバッグ

第2章

はじめて読むプログラム

いよいよ、C言語征服への登山に第一歩を踏み出します。本章では、C言語の裾野にあたる、プログラムのおおまかな構造と、基本的な概念について解説します。

はじめてプログラムを目にした時には、まるで暗号のようにしか見えないかもしれませんが、本章を読むうちに、はっきりと輪郭をつかむことができるようになります。遠くから見ただけでは、どこも同じように見えた山の斜面も、近づいてみれば表面には岩肌や樹木があることがわかるというわけです。

2.1

プログラムの構造

電卓による計算

最初の例題プログラムは、ごく身近で簡単な時間の計算です。プログラムの書き方を解説する前に、計算手順を電卓を使って解説しておきましょう。そうすれば、プログラムの処理内容がとてもわかりやすくなるからです。

例題では、現在の時刻を9時45分として、その1時間25分後には何時何分になるかを計算することにします。どこかへでかける時には必ず必要になる計算です。暗算でパッと計算できる人もいるでしょうが、ここでは次のような計算方法を考えてください。

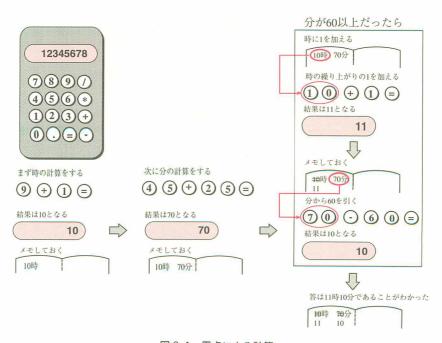



図 2-1　電卓による計算

まず、時の部分を足し算します。そして次に、分の部分を足し算します。ここで、分の部分が60以上だったら、60を引いて時の方に1繰り上げます。

この計算を電卓で行う様子を図に表したのが図2-1です。途中で計算結果をメモに記録しておくところなどは、重要なポイントですからこの計算手順をしっかり把握しておいてください。

コンピュータプログラムによる計算

では次に、同じ計算を行うプログラムを図2-2に示します。詳しくはこのあと解説していきますので、個々の計算式などを理解する必要はありませんが、注釈を読んで前の図との対応をつかんでください。

```
#include <stdio.h>

main()
{

    int hour,minute;

    hour = 9 + 1;
    minute = 45 + 25;

    if (minute >= 60)  {

        hour = hour+1;
        minute = minute-60;

    }

    printf("%d:%d\n",hour,minute);
}
```

時の加算を計算する
分の加算を計算する
分の計算結果が60以上ならば
時を1時間繰り上げ
分を60分戻す
計算結果を表示する

図2-2　コンピュータプログラムによる計算

プログラム実行の仕組み

コンピュータのプログラムは、図2-2のように数式に似た命令文を並べたものです。それぞれの命令文が電卓のキーをたたいたり、紙にメモするといった計算手順に相当します。コンピュータはプログラムに書かれた命令文を1つずつ順番に実行していきます。

この仕組みは、私たちが料理を作る様子とよく似ています。本を見ながら料理を作ることを考えてください。料理の本にはいろいろな料理の作り方が書いてありますが、これをレシピと呼びます。各レシピには図2-3のように、

調理手順が順を追って書いてあります。私たちはそれを読みながら材料を調理していきます。

コンピュータがプログラムを実行する仕組みもこれと同じようなものです。プログラムとして書かれた情報の処理手順を、コンピュータはひとつひとつ順番に実行していくのです。

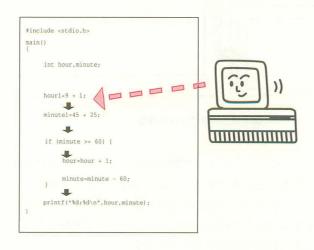



図 2-3 プログラム実行の仕組み

プログラムの構造

コンピュータにとってのレシピである、プログラムの書き方を見ていきましょう。図2-4は、レシピとプログラムの形式を比べたものです。図でわかるようにレシピには一定の形式があります。それは材料の一覧を記した部分と調理手順を記した部分に分けて書くことです。

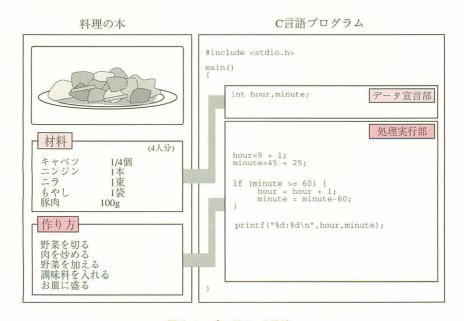



図2-4　プログラムの構造

プログラムにも同じように、決まった形式があります。宣言部は情報の種類や数を書き並べておくところです。レシピでは材料を書き並べることにあたります。ただし、プログラムの場合はどちらかというと鍋やフライパンのような調理道具を用意することに近いでしょう。材料となる情報を入れて、煮たり焼いたりする入れ物を用意するところです。

処理部は情報の処理手順を書き並べるところです。料理で言えば、フライパンに肉を入れて強火で焼く、といった調理手順を書き並べることにあたります。

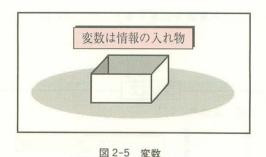
2.2

変　　数

プログラムのおおまかな構造がわかったところで、各部分の詳しい解説に入りましょう。このあたりからC言語の文法用語が出てきますが、決して難しくありませんから心配はいりません。文法といってもごく簡単な形式に従って書くだけのことです。

変数とは

宣言部では情報の入れ物を用意すると解説しました。文法的にはこの入れ物のことを**変数**（へんすう）と呼びます。変数という言葉には数学の方程式で使われる変数のイメージがあるかもしれませんが、プログラムにおける変数は単純に『情報の入れ物』を意味します。図2-5のような箱をイメージするとよいでしょう。

図2-5　変数

変数宣言

変数には、下の図2-6のように名前を付けておきます。変数名は自由に決めることができますが、なるべく中に入れる情報の種類を表す名前を付けるようにしましょう(変数名の付け方に関しては40ページのコラムを参照してください)。

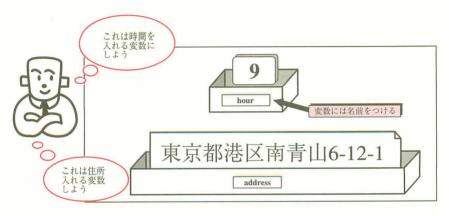



図2-6　変数名

変数を用意することを**変数を宣言する**といい、次の図2-7のような書式で書きます。

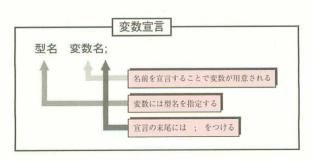



図2-7　変数宣言の書式

変数を用意するときには、名前だけでなく**型**を指定しなければなりません。型というのは、変数に入れる情報の種類を表すものです。コンピュータは、いろいろな種類の情報を扱うことができますが、どの種類を扱うかをあらかじめ指定しておかなければならないのです。

変数の型は、**型名**で指定します。例題のプログラムでは、最も基本的な型である**int（イント）型**を使います。intはinteger（インテジャー）からきた用語で、『整数』という意味です*1。そのほかの型については後の章で解説していきます。

例題のプログラムでは、図2-8のようにhourやminuteといった変数を宣言しています。宣言文にはかならず末尾に「;」（セミコロン）を付けることを忘れないでください。

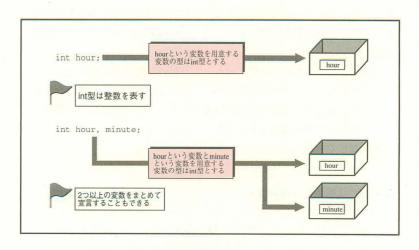



図2-8　変数宣言

＊1　整数とは0、1、2…のように、小数点以下の部分を持たない数のことです。

変数名の付け方

変数名は、次の簡単なルールさえ守れば自由に決めることができます。

■「a～z」、「A～Z」、「0～9」、「_」の文字だけを使う

■先頭の文字は数字以外の文字

■キーワードは除く

たとえば、「a」や「b」など1文字の変数名や「apple」や「orange」のような英単語、数字を使って「a1」「a2」「a3」や「part1」「part2」のような変数名を変数に付けることができます。大文字と小文字は区別されますので注意してください。たとえば、「Apple」と「apple」は違う名前として扱われます。

キーワードというのは、命令語などのことで、「int」や「if」のような単語です。キーワードとしてどのようなものがあるかは、本書を読み進むうちにだんだんわかってくるでしょう（Appendix6 キーワード一覧を参照）。

変数名のルールはこれだけですが、これ以外に慣習的に使われている命名法があります。その命名法を知っていると、ほかの人が書いたプログラムを読むときに理解の助けになるでしょう。具体的には、本書のプログラムをいくつか読むうちに少しずつわかってくると思いますが、後でまた詳しく解説します。

C言語の変数として有効な名前	C言語の変数として使えない名前
a　b　i　j	int………………キーワードの1つ
ch　ptr	MS-DOS………「-」は使えない
character　pointer	get_many_$…「$」は使えない
an_apple	1of2……………数字ではじまっている
number1　number2	

2.3

実行処理

変数宣言は、情報の入れ物を用意するための、いわばプログラム実行の準備段階です。計算などの処理は、**処理部**に書かなければなりません。

次は、その処理部について解説しましょう。

式と代入

図2-9aは時間を計算するプログラムの一部で、「9+1」を計算してその結果を変数 hour に入れるところです。変数に入れることは、電卓の計算でいえば紙にメモすることにあたります。

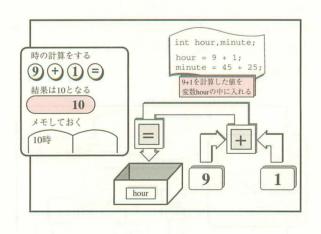



図 2-9a　代入(その 1)

変数に値を入れることを代入と呼びます。「＝」（イコール）記号は代入を指示する一種の命令です（図2-9b参照）。

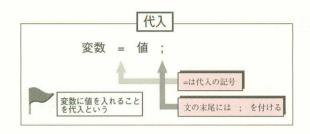



図2-9b　代入の書式

図2-10は、やはり時間計算プログラムの一部ですが、＝記号を等号と考えると、「hour」と「hour＋1」が等しい、というおかしな式になってしまいます。しかし、C言語では＝記号は代入を指示する命令なので、ここでは「hour＋1」を変数hourに代入する、という意味になります。

この例でわかるように、同じ変数に何度でも値を代入することができます。このとき、変数の値は新しい値に置き換えられ、それまで入っていた値は捨てられてしまいます。

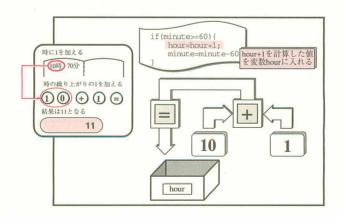



図2-10　代入（その2）

文

図2-11に示すように、C言語のプログラムは「文」を単位として実行が進められます。1つの文を実行したら次の文へ進むというように、順番に文を実行していきます。プログラムは、小さな処理を行う文を並べて全体として大きな処理を行うものなのです。

なお1つ1つの文の末尾にかならず「；」を付けます。慣れないうちは忘れやすいので注意してください。

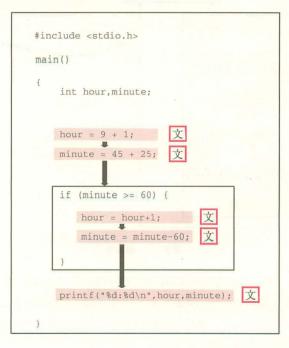



図2-11　文

条件分岐

時間を計算する場合、分の計算結果が60以上ならば、繰り上がりの処理をしなければなりません。図2-12に示すように「if (minute>= 60)」という部分が、『分が60以上ならば』を意味しています。このような文を**条件文**と呼びます。

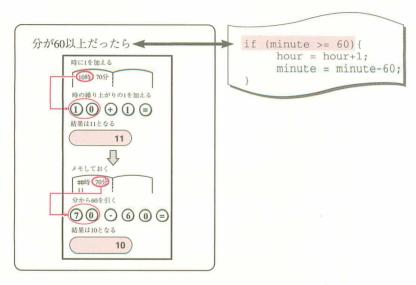



図2-12　条件文

「minute>= 60」は、変数minuteの値が60以上ならば真になり、60未満ならば偽となる**条件式**です。図2-13に示すように、■の部分はこの条件式が真のときしか実行されず、条件式が偽のときにはこの部分を実行せずに、そのまま次の処理に進みます。

条件にあてはまるかどうかによって処理の流れが2つに分かれるので、このような処理を**条件分岐**と呼びます。

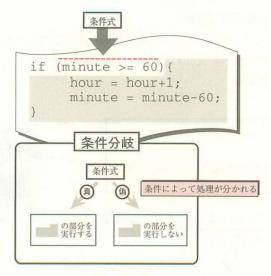



図 2-13　条件分岐

ブロック

条件分岐の処理では、条件式が真のときに2つ以上の文を実行しなければならないことがあります。例題プログラムでも、2つの文を実行しなければなりません。

このような場合には、図2-14のように2つの文を「{ }」(中カッコ)で囲みます。複数の文を「{ }」で囲ったものを**ブロック**と呼びます。

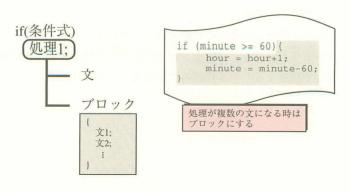
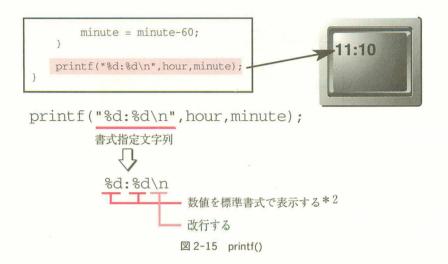



図 2-14　ブロック

プログラム部品の組込み

図2-15は、章末の例題プログラム中の画面に計算結果を表示する部分です。画面に文字や数字を表示するために、「printf」（プリントエフ）という命令を使っています。

図2-15　printf()

実はこの「printf」は、C言語の命令というわけではなく、プログラム部品の1つです。C言語では図2-16のように、プログラムの中にプログラム部品を組み込むことができます。図2-15では、「printf」という部品をプログラムに組み込んでいるのです。

C言語自身は画面表示などの機能を持っておらず、その代わり、こうした機能はプログラム部品として提供されています。それを、自分のプログラムに組み込んで利用するという仕組みになっているのです。

＊2　printfの書式については、Appendix2を参照してください。

プログラム

```
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1;
    minute = 45 + 25;

    if (minute >= 60) {
        hour = hour + 1;
        minute = minute-60;
    }
    printf("%d:%d\n",hour,minute);
}
```

プログラム部品集

putcher()	画面に文字を表示する
puts()	画面に文字列を表示する
printf()	書式指定付きで画面に文字列を表示する
⋮	
getcher()	キーボードから1文字入力する
gets()	キーボードから文字列を入力する
⋮	
fgetc()	ファイルから1文字読み込む
fputc()	ファイルに1文字書き込む

プログラム部品を組み込むことによって，画面表示などを行う

図 2-16　プログラム部品

プログラム部品のための命令

34 ページの図 2-2 に示したように、例題プログラムの中でこれまで解説してきた部分は、電卓での計算にそのまま対応します。逆にいうと、それ以外の部分は計算そのものには対応しない、つまり実行されない部分ということになります。実は、この部分はプログラム部品を作るための命令です。C言語では、プログラムを部品の集まりとして作成します。このため、プログラムを部品にする命令やほかの部品を利用するための命令を書いておかなければならないのです（図 2-17)。

```
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1;
    minute = 45 + 25;

    if (minute >= 60) {
        hour = hour+1;
        minute = minute-60;
    }

    printf("%d:%d\n",hour,minute);
}
```

プログラム部品化のための命令

図 2-17　処理に対応しない命令

プログラム部品については後の章で詳しく解説しますが、本章では例題のプログラムを実行させるために最低限必要な命令を簡単に紹介します。ここでは意味がよくわからないかもしれませんが、あまり気にせず、おまじないのつもりで書くようにしてください。

図2-18のように、プログラムの先頭の方に「#include　<stdio.h>」という行があります。#include文については7章で詳しく解説しますが、「printf」などのプログラム部品を利用するために必要なものです。

```
#include <stdio.h>
main()
{
```

プログラム部品printf()を利用するための命令

図2-18　#include文

図2-19はプログラム全体がブロックであること、つまり「{ }」で囲まれていることを表しています。これはプログラム部品の範囲を示すブロックです。

main( )というのはプログラム部品の名前を表しており、ここでは「main( )」という名前のプログラム部品を作成したことになります。なお、このプログラムでは、この部品の名前はかならず「main( )」でなければなりません。

```
#include <stdio.h>
main()
{
    int hour,minute;

    hour = 9 + 1;
    minute = 45 + 25;

    if (minute >= 60) {
        hour = hour+1;
        minute = minute-60;
    }

    printf("%d:%d\n",hour,minute);
}
```

プログラム全体をブロックにし、main()という名前のプログラム部品にしている

図2-19　main()

2.4

プログラムのスタイル

プログラムのスタイルとは

プログラムの書き方は、文法上定められている**書式**と、プログラマーの流儀にまかされている**スタイル**の2つに大きく分けることができます。

書式は記号や式を書く順序を定めているもので、かならず守らなければなりません。これに対して、スタイルは図2-20のように単語や記号の配置のことで、とくに決まりはなく自由に書くことができます。

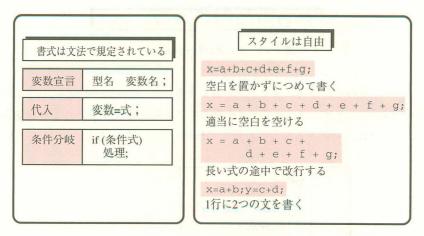



図2-20　プログラムのスタイル

図のように、式の途中で改行したり、1つの行に複数の文を書いてもかまいません。その代わり、文の末尾に「；」を書くことは忘れないでください。C言語では、スタイルを自由にする代わりに、「;」で文の終わりを明示するようにしているのです。

インデンテーション

スタイルに決まりはありませんが、1つだけ守ってほしいことがあります。それは**インデンテーション**を行うことです。インデンテーションとは、図2-21に示したようにブロック内やifの処理文を右に数文字分ずらして文頭を揃えることで、「段付け」や「インデント」ともいいます。

ずらす文字数をインデント幅といいますが、通常はタブの幅だけずらします。エディタによってはタブ幅を変更できるものもあるので、自分の好きな幅に調節するとよいでしょう。

```
if (minute >= 60) {
    hour = hour+1;
    minute = minute-60;
}
```

文頭をずらすことを インデンテーション という

図2-21　インデンテーション

ブロックの中にさらにブロックを書くときには、図2-22のようにもう1段深く段付けします。こうすると、if文の中にさらにif文を書いたような場合に、どのifに対応する文なのかがすぐわかるようになります。

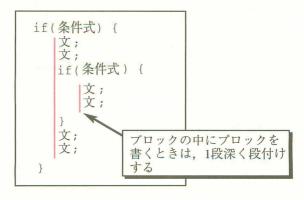



図2-22　インデンテーションの例

ブロックのインデンテーションにはいくつかの流儀があります。その一例を図2-23に示しましょう。本書では(A)の流儀で書いていますが、みなさんは自分の気にいった流儀を選んで使ってかまいません。

(A)
```
if (minute >= 60) {
        hour = hour+1;
        minute = minute-60;
}
```

(B)
```
if (minute >= 60)
{
        hour = hour+1;
        minute = minute-60;
}
```

(C)
```
if (minute >= 60)
    {
        hour = hour+1;
        minute = minute-60;
    }
```

ブロックのインデンテーションにはいくつかの流儀がある

図2-23 ブロックのインデンテーション

コメント

[書式] /* [コメント] */

図2-24は例題プログラムに**コメント**を加えたものです。コメントは『注釈』という意味で、プログラムにメモとして書き込む解説文のことです。図2-24に示すように、「/*」と「*/」で囲んだ部分はすべてコメントになります。図のように、コメントはどこにでも自由に書くことができます。

```
/*このプログラムは時間の加算を行なうプログラムです*/
/* printf()を使うための命令 */
#include <stdio.h>
main()
{
/* 変数宣言部 */
    int hour,minute;
/* ここから実行処理部 */
    /* 加算処理 */
    hour = 9 + 1;         /* 時 の計算 */
    minute = 45 + 25;     /* 分 の計算 */
    /* 操り上がりの処理 */
    if (minute >= 60) {           /* 分 が60以上なら */
        hour = hour+1;            /* 時 を操り上げ*/
        minute = minute-60;       /* 分 を60戻す */
    }
    /* 計算結果を画面に表示する */
    printf("%d:%d\n",hour,minute);
}
```

図 2-24　コメント

コメントは、プログラムを読む「人間」のために書いておくもので、コンピュータはコメントを読み飛ばしてプログラムを実行します。

コメントには処理内容を解説する文を書いておきます。たとえば、「hour = 9+1;」という文は時間計算の『時の部分の計算』を行っているところですが、このプログラムをはじめて読む人にとってはコメントがなければ何の計算だかわからないでしょう。また、たとえ自分で書いたプログラムであっても、日数が経ってから読み返すと忘れてしまって、わからなくなるものです。ほかの人にも利用してもらうプログラムはもちろんですが、自分で利用するプログラムでも、思いどおりに動いてくれないときや改造したくなったときなどには、かならず読み返すことになります。わかりやすい変数名と、わかりやすいコメントを付けるように習慣をつけておきましょう。

なお、本書の例題プログラムでは紙面の都合上コメントを付けませんが、みなさんがプログラムを入力する場合には、コメントを付けるようにしてください。

本書では、このあとC言語の文法を詳しく解説していきますが、プログラムのスタイルについてはこれ以上言及しません。みなさんがプログラムを作成する際には、図2-25のように、プログラムが読みやすくなるスタイルで適宜工夫して書くようにしてください。演算子の前後に空白を入れたり、処理のまとまりごとに空白行を入れるのも1つの方法です。

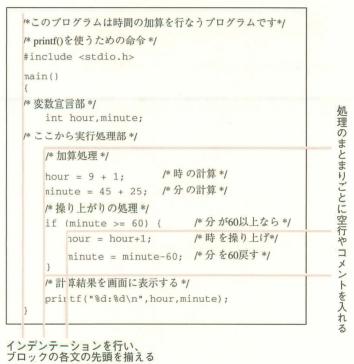



図2-25 プログラムスタイルの例

〈本章で取り上げたプログラム〉　プログラム2-1

```
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1; ………………………………………時の加算を計算する
    minute = 45 + 25; …………………………………分の加算を計算する

    if (minute >= 60) { ……………………………分の結果が 60 以上ならば
        hour = hour+1;……………………………………時を 1 時間繰り上げ
        minute = minute-60;……………………………分を 60 分戻す
    }

    printf("%d:%d\n",hour,minute); ………………計算結果を表示する
}
```

第3章

C言語の3つの柱

“

下の表は、本書の全体構成図の一部です。C言語の各機能を役割によって分類すると、この表に示すように3つの柱があることがわかります。本章の目標は、1つ1つの柱が担っている役割をつかむことにあります。

C言語の習得で重要なのは、この3つの側面のそれぞれについて、初歩から順序よく学習していくことです。ひとつの側面だけ詳しく学習しても、うまく使いこなすことはできません。初歩的な機能しか習得していなくても、3つの側面についてバランスよく知っている方が幅広く応用できるのです。3つの側面をバランスよく使いこなして、C言語を確実に身につけるようにしてください。

表はC言語の全体像と本書の解説の関連を示したものです。このように、C言語の機能は、大きく3つに分けられます。本章では、各機能の役割を大まかに解説します。

C言語修得のポイントは、この3つの機能をバランスよく学習していくことでしょう。1つの機能だけを詳しく学習しても他の機能との連係がなければ使いこなすことはできません。

”

本章で解説する項目

	データ型	部品化機能	制御構造
3.1			実行の流れを変える 繰り返し
3.2	基本データ型 複合データ型 char型 特殊文字		
3.3		関数定義 部品化と再利用 ライブラリ関数	
		関数実行の仕組み	

3.1 演算と実行の流れ

3.1.1 文

文で行う処理

プログラムにおける処理の最小単位は「文」です。コンピュータは、文を1つ1つ順番に実行して処理を進めていきます。

文で行う処理のほとんどは、図3-1のように、非常に簡単な計算かプログラム部品の呼び出しばかりです。計算途中の値をメモに書き留めるように、途中結果を変数に格納しながら計算を進めます。そして、最終的な計算結果を画面に表示するなどして出力します。

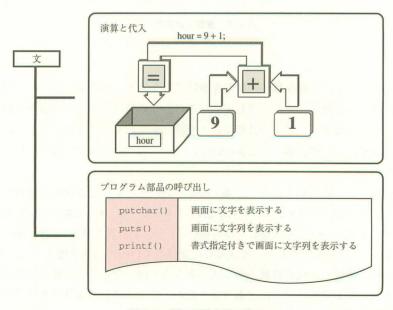



図3-1　文は処理の最小単位

演算と演算子

計算式の中で使う「＋」記号や「－」記号は、足し算や引き算という個々の**演算**を指示する一種の命令です。このような記号のことを**演算子**（えんざんし）と呼びます。「＝」記号も、代入という演算を指示する演算子です。

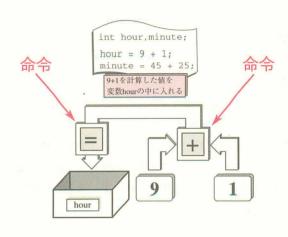



図 3-2　演算子は命令

次の表3-1に主な演算子の種類を挙げておきます。掛け算には「*」、割り算には「/」を使うことに注意してください。この他にも多くの演算子を使うことができますが、通常の計算ではこの図に挙げたもので十分でしょう。他の演算子については巻末のAppendix3に挙げておきますので、参考にしてください。

演算子には優先順位があり、優先順位の高いものから順に演算が実行されます。例えば「1+4＊2」という式では、「＋」よりも「*」の優先順位が高いので、まず「4＊2」を計算してから次に「1+8」を計算します。「(1+4)＊2」のように「( )」を付ければ、優先順位に関係なく( )内の演算が優先されます。

四則演算は小学校の算数で習う式の書き方と同じなので、演算子の優先順位を気にする必要はほとんどありませんが、C言語特有の演算子を使う場合には優先順位に気を付けてください。

優先順位	演算子	意味	使用例	
高い ↑	*	乗算	a = b * c;	b と c を掛けた値を a に代入
	/	除算	a = b / c;	b を c で割った値を a に代入
	%	剰余算	a = b % c;	b を c で割った余りを a に代入
↓ 低い 高い ↑	+	加算	a = b + c;	b と c を加えた値を a に代入
	-	減算	a = b - c;	b から c を引いた値を a に代入
↓ 低い	=	代入	a = b;	b を a に代入

表 3-1　主な演算子

変数の値を変更する演算子

変数に新しい値を格納する場合には、代入演算子「=」を使います。ただし、42ページの図2-10のように、変数のそれまでの値を利用して新しい値を計算する場合には、次の表3-2のような演算子を使うことができます。

優先順位	演算子	意味	使用例
高い ↑	++	インクリメント	a++;
	--	デクリメント	a-- ;
↓ 低い	=	右辺を左辺に代入	a = b;
	*=	左辺と右辺を乗算し、左辺に代入	a *= b;(a = a * b;と同じ)
	/=	左辺を右辺で除算し、左辺に代入	a /= b;(a = a/b;と同じ)
	%=	左辺を右辺で剰余し、左辺に代入	a %= b;(a = a%b;と同じ)
	+=	左辺と右辺を加算し、左辺に代入	a += b;(a = a+b;と同じ)
	-=	左辺を右辺で減算し、左辺に代入	a -= b;(a = a-b;と同じ)

表 3-2　変数の値を変更する演算子

次の図3-3は、++演算子や-=演算子を使って前章の例題プログラムを書き直したものです。慣れないうちは奇妙な式に見えるかもしれませんが、C言語のプログラムではこちらの方が一般的です。慣れるとプログラムの意味がわかりやすくなり、しかも5章で解説するように実行速度の速いプログラムになるからです。

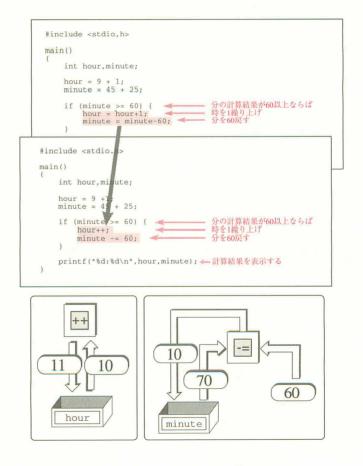



図3-3　++演算子と-=演算子

++演算子を使って変数の値を1つ増やすことを、変数を**インクリメントする**といいます。逆に、--演算子を使って変数の値を1つ減らすことを変数

を**デクリメントする**といいます。コンピュータのプログラムでは、変数の値を1つずつ変化させる処理が多いので覚えておくとよいでしょう。

3.1.2　プログラム実行の制御のための構文

実行の流れを変える

プログラム実行の基本は、下の図3-4のような**直線的な流れ**です。プログラムを上から下までまっすぐ実行するので、『直線的』というわけです。この直線的な流れを変えることによって、プログラムで実現できる処理にバラエティを持たせることができます。C言語にはプログラムの流れを変える方法として、図3-4のような**条件分岐**と**繰り返し**の2つが用意されています。

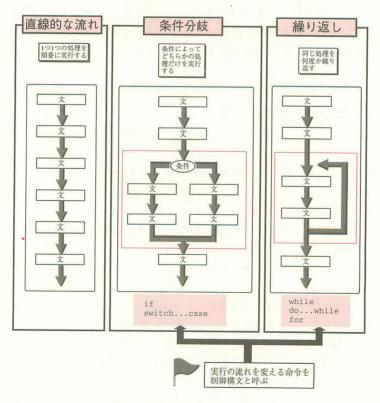



図3-4　プログラム実行の流れ

「条件分岐」は、2章で解説したようにプログラム実行の流れを条件によって振り分ける方法です。入力されたデータの内容により異なる処理をしたり、演算した結果により処理を選択するような場合に利用します。条件分岐の構文には図3-4に示すように **if 文**と **switch 文**があります。

「繰り返し」は、同じ処理を何度も実行する方法です。ある処理のためにデータがすべて入力されるまで繰り返したり、条件にあてはまるものを見つけるまで繰り返したりする場合に利用します。繰り返し処理の構文には、図3-4に示したように **while 文**、**do～while 文**、**for 文**の3つがあります。

C言語には、プログラムの実行の流れを変えるために全部で5つの構文が用意されていますが、基本的には条件分岐と繰り返しの2種類しかありません*1。しかし、たった2種類の構文でも、その表現力は豊かです。私たちが日常行っている行動も、つきつめれば、ほとんどの場合は条件分岐と繰り返しで表現することができるのではないでしょうか。本節では例題プログラムを作成しながら、繰り返し処理の構文を解説します。

ノイマン型コンピュータ

コンピュータは、年を追うごとにどんどん高速化されており、パーソナルコンピュータでも1秒間に数百万回もの命令を実行できるようになりました。コンピュータの高速化はこれからも進むでしょう。

現在のコンピュータの多くは、「直線的な実行」を基本にしています。条件分岐や繰り返しで実行の流れをコントロールすることはできますが、ひとつひとつの命令を順次実行していることに変わりはありません。このようなコンピュータの仕組みのことを、コンピュータの誕生に大きく貢献したフォン・ノイマン氏の名前をとって「ノイマン型コンピュータ」と呼んでいます。

ノイマン型コンピュータを高速化するには、ひとつひとつの命令の実行を高速化するしかありません。ところが、コンピュータが1命令を実行する間に、光でさえも数10m程度しか進まないほど高速化されてくると、それ以上高速化するのは非常に困難となってきました。

＊1　このほか goto 文がありますが、本書では扱いません。

このため「直線的な実行」だけではなく、なんらかの形で「並列実行」、つまり2つ以上の命令を同時に実行する機能が高速化のために不可欠の技術となっています。

繰り返し

繰り返し処理は同じところをぐるぐる回るという意味で、**ループ処理**とも呼ばれます。ループというのは輪のことで、繰り返す処理を1回実行することを、「ループを1回まわる」という呼び方をします。

繰り返し処理の構文には、図3-5のような3つの種類があります。各構文の違いも示しておきましたので参考にしてください。

繰り返し処理

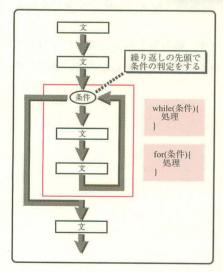
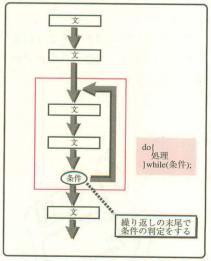
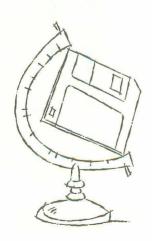



図3-5　繰り返しの構文

「for 文」と「do～while 文」については4章で詳しく解説することにして、ここでは while 文を解説します。「while」は『～の間』という意味で、( )内の条件が満たされている間、指定された処理を繰り返すというものです。繰り返し処理ではループを1回まわるたびに条件がチェックされ、満たされていなければその時点で繰り返し処理を終了して、次の処理に進みます。

カセットテープ交換時刻表示プログラム

それでは「while 文」を使ったプログラムを作成してみましょう。例題のプログラムは、次のようなものです。ある会議の様子をカセットテープレコーダで録音することを考えてください。会議は数時間に及ぶので、途中で何回かカセットテープを裏返したり、取り替えたりしなければなりません。そのためのカセットテープを交換する時刻を、あらかじめ計算しておくプログラムです。60分テープならば簡単な計算で済むのですが、手元に46分テープしかないので23分おきにカセットテープの交換をしなければならないとしましょう。

プログラムは、図3-6のようになります。繰り返し処理の部分に注目してください。計算の内容は3章のプログラムとまったく同じなので、理解しやすいと思います。

```
#include <stdio.h>

main()
{
    int hour,minute; ……………………………… 時刻を入れる変数
    int tape_len,tape_num; ………………………… テープの数を入れる変数
                 ……………………………………… テープの長さを入れる変数

    hour   = 13; }
    minute = 30; } ………………………………… スタート時刻を 13:30 にする

    tape_len = 46/2; …………………………………… テープの長さは片面ごとなので 46÷2
    tape_num = 3*2; ……………………………………… テープの数×2 が交換する回数
    while (tape_num >0 ) { ………………………… テープの数が正の間繰り返す
        tape_num--; ………………………………………… テープの数を 1 つ減らす

        printf("%02d:%02d にテープを交換してください\n",hour,minute);
                              ……………………… 時刻を表示する
        minute = minute + tape_len; ………………… テープの長さを加算する
        if (minute >= 60) { ………………………………… 分の結果が 60 以上ならば
            hour++; ………………………………………………… 時を 1 時間繰り上げ
            minute -= 60; ……………………………………… 分を 60 分戻す
        }
    }
}
```

＜実行結果＞

```
13:30にテープを交換してください
13:53にテープを交換してください
14:16にテープを交換してください
14:39にテープを交換してください
15:02にテープを交換してください
15:25にテープを交換してください
```

図 3-6　カセットテープ交換時刻表示プログラム

3.2

データ型

3.2.1　データ型とは

基本データ型

変数を使って情報を処理する仕組みは、コンピュータ内部で情報を処理する仕組みにそのまま対応させたものです。驚くべきことですが、コンピュータ内部で処理できる情報は図3-7に示すようにほんの数種類しかありません。

C言語では、変数に対応する情報の種類を**データ型**として区別します。これら図3-7に示したデータ型は、コンピュータ内部の仕組みにそのまま対応したデータ型という意味で、**基本データ型**と呼びます。

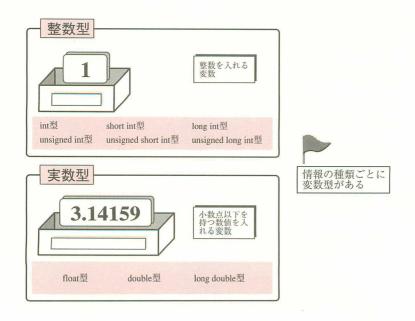
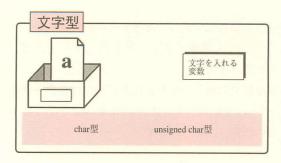
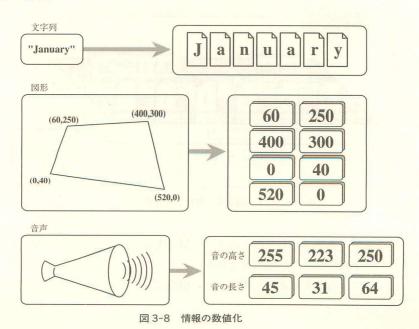



図 3-7　基本データ型

この図では同じ情報の種類にいくつものデータ型が対応していますが、これについては 6章で詳しく解説します。

複合データ型

数値の計算を行うプログラムでは、基本データ型の変数を利用することができますが、数値や文字以外の情報を扱う場合にはそのままでは処理できません。そこで、図 3-8のように情報を数値や文字に置き換えて処理することになります。

図 3-8　情報の数値化

この図のように、情報を数値の組で表して処理するのが、コンピュータの大きな特徴です。どんな情報でも数値や文字に置き換えてしまえば、コンピュータで処理することができます。

C言語では、このような数値の組として表される情報を**複合データ型**の変数として扱います。複合データ型の変数は、次の図3-9のように、基本データ型を組み合わせて1つの変数にしたものです。

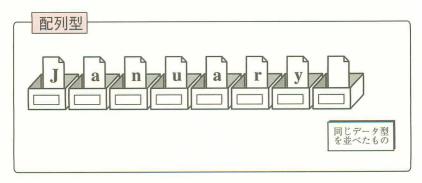
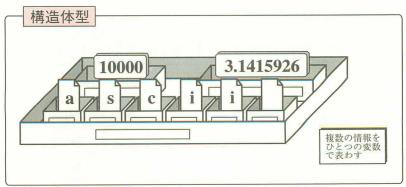



図3-9　複合データ型

複合データ型は、あらかじめデータ型が用意されているのではなく、データ型を作る仕組みとして用意されています。プログラマー自身が基本データ型の組み合わせを考えて、処理したい情報にふさわしい新しいデータ型を作るのです。

なお、この他にポインタ型の変数がありますが、これについても後の章で詳しく解説します。

3.2.2 char型

基本データ型のひとつである、文字型を扱うプログラムを作成してみましょう。次の図3-10は、前節の例題プログラムのメッセージを表示する部分を改造したものです。

```
#include <stdio.h>

main()
{
    char c;                          ……入力した文字を入れる変数
    int hour, minute;                ……時刻を入れる変数
    int tape_len, tape_num;          ……テープの数を入れる変数
                                     ……テープの長さを入れる変数
    hure=13;     }
    minute=30;   }                   ……スタート時刻を13:30にする

    tape_len = 46/2;                 ……テープの長さは片面ごとなので46÷2
    tape_num = 3*2;                  ……テープの数×2が交換する回数

    while (tape_num>0) {             ……テープの数が正の間くり返す
        tape_num--;                  ……テープの数を1つ減らす

        do {
            c = getchar();           ……キーボードから1文字入力する
            if (c=='?')              ……文字が「?」だったら時刻を表示
                printf("%2d:%02d",hour,minute);
            else                     ……「?」でなければ
                putchar(c);          ……文字をそのまま表示
        } while (c!='\n');           ……改行キーが押されるまでくり返す

        minute = minute + tape_len;  }
        if (minute >= 60) {          }……前のプログラムと同じ
            hour++;                  }  時間の加算を行なう
            minute-=60;              }
        }
    }
}
```

図3-10 交代時刻表示プログラム

このプログラムの実行例を次ページの図3-11に示します。このプログラムは、カセットテープを交換するときに書記を交代することにして、その時刻を表示するためのものです。

実行例を見てわかるように、キーボードからメッセージを入力し、それをそのまま出力しますが、文字「?」だけは時刻に置き換えて表示します。

プログラムを実行し，出力を
ファイルturn.outにリダイレクトする

```
A>turn > turn.out
From ? ,Mr.Suzuki's turn.
From ? ,Mr.Minami's turn.
From ? ,Miss Chieko's turn.
Form ? ,Mr.Yamada's turn.
From ? ,Mrs.Yuko's turn.
From ? ,Mr.Sekimoto's turn.
A>type turn.out
From 13:30 ,Mr.Suzuki's turn.
From 13:53 ,Mr.Minami's turn.
From 14:16 ,Miss Chieko's turn.
From 14:39 ,Mr.Yamada's turn.
From 15:02 ,Mrs.Yuko's trun.
From 15:25 ,Mr.Sekimoto's turn.
A>
```

キーボードから入力する

プログラムの出力

図 3-11　交代時刻表示プログラムの実行例

このプログラムでは、文字を処理するためにchar型の変数を使っています。char型の変数には図3-12のように文字を格納することができます。

```
main()
{
    char c;
    int hour,minute;
    int tape_len,tape_num;
```

F

c

図 3-12　char 型変数

図3-13に示した「getchar」（ゲットチャーと読む）はプログラム部品の一種で、図のように、getcharを呼び出すたびにキーボードから入力された文字を1文字ずつ順番に返します。「putchar」（プットチャーと読む）もプログラム部品の1つで、指定した1文字を画面に表示します。

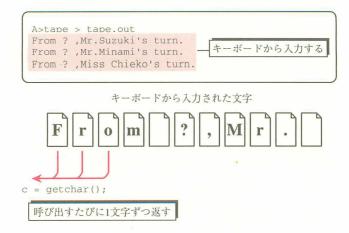



図 3-13 getchar

実行例では実行結果がわかりやすいように、画面への表示をいったんファイルに格納しておき、後であらためて表示しています。

図 3-14 は、getchar で入力した文字が「?」かどうかを調べる部分です。図のように、プログラム中で「文字という情報」を表すには、目的の文字を「'」（シングルクォーテーションマーク）で囲みます。

```
do {
    c = getchar();
    if (c=='?')
        printf("%2d:%02d",hour,minute);
    else
        putchar(c);
} while (c!='\n');
```

文字 ? を、プログラムでは '?' と書く

図 3-14 文字

なお、このプログラムで使っている do～while 文は、while 文と同じ繰り返しの構文です。詳しくは後で解説します。

if 文の後の else の次の文は、if の条件が成立しない場合に実行される文で

す。cが「?」と等しいかどうかを調べるには、「=」ではなく「==」を使います。また、do~while文の条件式のように「\n」と等しくないかどうかを調べるには、≠ではなく「!=」を使います。

特殊文字

コンピュータのキーを押すと、それに対応する文字がコンピュータに入力されます。getcharはさきほどの図3-13のように、入力された文字を1文字ずつ返します。リターンキーは文字には対応しませんが、リターンキーが押されるとgetcharは、図3-15のように特殊な文字を返します。

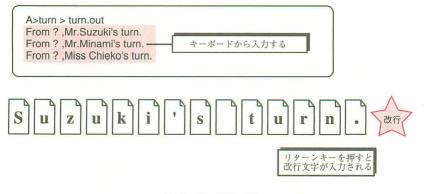



図3-15　特殊文字

C言語では、図3-16のようにリターンキーに対応する特殊な文字を「\n」と表します。

```
do {
    c = getchar();
    if (c=='?')
        printf("%2d:%02d",hour,minute);
    else
        putchar(c);
} while (c!='\n');
```

文字 改行 を、プログラムでは'\n'と書く

図3-16　\n

「A」や「B」などの文字をputcharで表示すれば、画面に文字が表示されます。ところが、リターンキーに対応する文字'\n'をputcharで表示すると、文字が表示される代わりに、カーソルが次の行の先頭に移動します。つまり、「改行」が実行されるのです。

このように、改行したり、画面をクリアしたりする処理を**画面制御**と呼びます。特殊文字を使って通常の文字を表示するのと同じ方法で、画面制御を行うことができるのです。

画面制御のための特殊文字の例を、次の図3-17に示します。C言語では、画面制御文字を表すために、「\」(バックスラッシュ)を使います*2。「\n」や「\f」のように「\」+「1文字」で画面制御文字を表すのです。

図3-17 画面制御文字

画面制御文字は、文字列の中にも置くことができます。例題のプログラムでもprintfのなかで、画面に表示する文字列の中に"...\n"という形で使っています。この文字列を表示すると最後に改行されますが、「\n」がなければ改行されません。

*2 日本のパーソナルコンピュータでは、ほとんどの機種で「\」の代わりに「¥」(円マーク)を使うようになっています。例題プログラムで「\」となっているところでは、「¥」を使ってください。

3.3

プログラムの部品化

プログラム建築学

C言語の特長の１つとして、プログラムの部品化機能を挙げることができます。建築にたとえると、プログラムの部品化機能があるのとないのとでは、〝ほら穴住居〟と〝木造住居〟くらいの大きな違いがあるといってよいでしょう。

原始時代、人類はほら穴に住んでいたと言われています。大きな岩や崖を

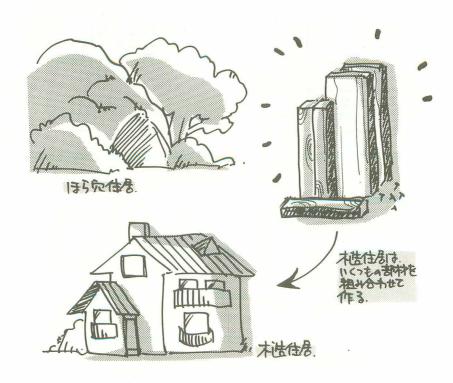



図 3-18　ほら穴住居と木造住居

くりぬいて穴を掘って使ったり、自然の洞窟を住居として利用したのでしょう。ほら穴住居の特徴として注目しておきたいのは、大きな岩というひとつの材料からできていることです。

やがて文明が芽生えると、人類は石や木材を使って住居を作るようになりました。日本では木や竹を使って家を建てます。木材を柱や板に加工して、それを組み立てて木造住居を作ります。ほら穴住居が1つの材料からできているのに対して、木造住居がいくつもの部材を組み合わせてできあがることに注目してください。これは、大きな違いです。

以降では、この考え方をC言語にあてはめて解説してみましょう。

関数とは

関数は、プログラムの一部を「部品」として分離したもので、木造住居では柱や床板に相当します。木材を切り出して柱や壁板を作るように、関数をプログラムに合せて作ることもできます。そして、柱や壁板を組み立てて家を建てるように、C言語では関数を組み合わせることによってプログラムを構築するのです。このことを表したのが図3-19です。

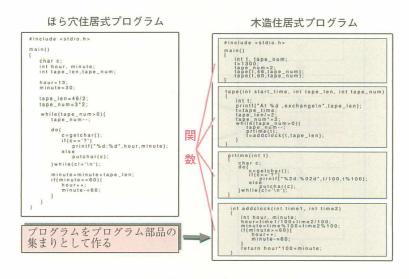



図3-19　関数

関数の定義

プログラムを関数にするための書式は、図3-20のように非常に簡単です。このように関数を作成することを**関数を定義する**といいます。

図3-20　関数定義

関数実行の仕組み

定義した関数をプログラム部品として呼び出すには、図3-21のように関数を実行したいところに関数名を書いておくだけです。関数はあたかもC言語にあらかじめ備わった命令文のように使うことができます。

関数の実行を指示することを、**関数を呼び出す**といいます。関数呼び出し

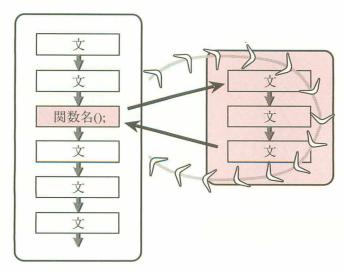



図3-21　関数の呼び出し

の仕組みは、まるでブーメランのようなものです。まず、部品として切り出した関数を呼び出すと、そこで実行を一時中断して、関数の先頭から実行をはじめます。そして、関数のプログラムの実行を終了すると、呼び出したところに舞い戻って元のプログラムの実行を再開するのです。

関数による処理の集中化

関数実行の仕組みを利用して、ひとつのプログラムの中で同じような処理を1か所にまとめることができます。図3-22のように、プログラムのあちらこちらで関数部品を呼び出すことによって、同じ処理を何か所でも実行することができるのです。

これは言ってみれば、木造住居で、1つの家に同じ寸法の柱が何本も使われているようなものです。

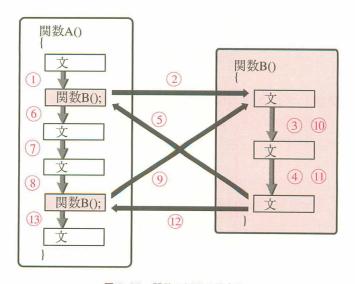



図 3-22　関数と処理の集中化

部品化と再利用

木造住居では、同じ寸法の柱や壁板などを他の家を作るときにも使うことができますが、プログラムでも同じことがいえます。図3-23のように、他のプログラムでも同じ関数を再利用できるのです。

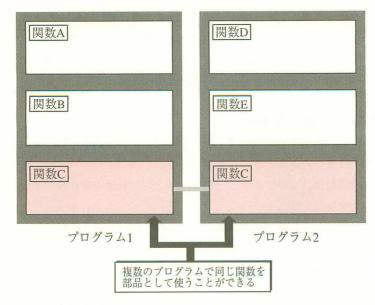



図3-23　関数と部品化

main()関数とプログラムの構造

前章までに作成したプログラムも、実は関数として作成されています。48ページの図2-19を見るとわかるように、プログラム全体がmainという関数になっています。

mainは，文字どおり『主要な』という意味で、特別な役割を持っています。それは**かならず最初に呼び出される関数**というものです。C言語で書かれたプログラムの実行が開始されると、図3-24のようにまず最初にmain()関数の先頭から実行されるのです。そして、main()関数の実行が終了すると、プログラムの実行も終了します。

関数による部品化機能をうまく利用すると、プログラムは、図3-24のようにmain()関数を頂点とする階層構造になります。処理の内容ごとに関数として部品化し、処理の流れをつかみやすくするのです。

ほら穴住居では、どこまでが玄関でどこからが廊下とはっきり区切ることはできませんが、木造住宅では、その区切りがはっきりしています。プログラムでも、処理手順がはっきり区切られているほうが構造をつかみやすくな

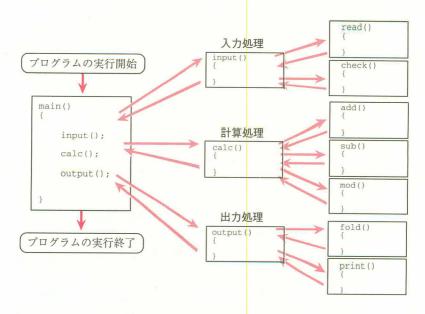



図 3-24　main()関数とプログラムの構造

ります。この図 3-24 のように、処理手順のひとつひとつを関数としてはっきり分離すると、プログラムの構成が一目で見渡せて理解しやすいのです。

プログラムの構造をわかりやすくすることは重要です。なぜなら、プログラムのメンテナンスにたいへん役立つからです。プログラムにミスがあっておかしな動作をする場合など、構造がはっきりしていれば誤動作の原因となる部分を特定しやすくなります。また、プログラムを改良したり部品を再利用したりするときにも好都合です。

みなさんも関数の機能をうまく使って、プログラムの構造をわかりやすく書くように習慣づけてください。

ライブラリ関数

C言語処理系には、**ライブラリ関数**と呼ばれる関数があらかじめ多数用意されています。ライブラリは『図書館』という意味で、ライブラリ関数は最初から用意されているプログラム部品といってよいでしょう。プログラムを作るときには、図 3-25 のようにライブラリ関数の中から必要な関数を自由に

呼び出して利用することができます。ライブラリ関数を呼び出すことで、画面への表示やファイルの読み書きなどの処理を行います。ライブラリ関数はC言語処理系のメーカーによって提供されている便利な部品集とでも思えばよいでしょう。

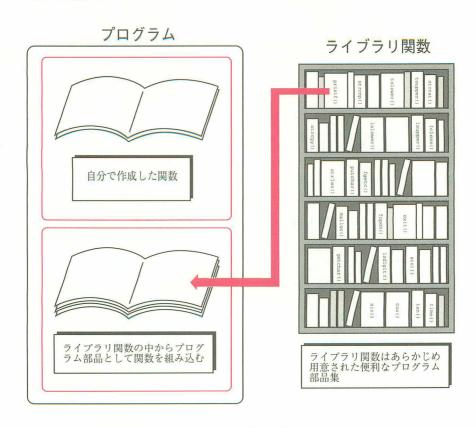



図 3-25　ライブラリ関数

関数はあくまで部品ですから、C言語の文法でライブラリ関数の種類が決められているわけではありません。しかし、ANSI規格ではプログラムの移植性を高めるために、文法とは別に基本的な部品集として統一されています。このような基本部品集に含まれている関数のことを**標準ライブラリ関数**と呼びます。たとえば、これまで画面に出力するために使っていたprintf()関数も

標準ライブラリ関数のひとつです。標準ライブラリ関数を使っていれば、どの処理系を使っていても、同じプログラムを実行することができます。

処理系によっては、標準関数以外にも多くのライブラリが用意されています。たとえば、パソコン用の処理系などではグラフィックスやサウンドを扱う関数が用意されているものもあります。

関数実行の仕組みと、部品化によるメリットを理解したところで、本章での解説はひとまず終わります。関数を使ったプログラミングの実際については次の章でじっくり解説しますので、少し休憩してください。

〈本章で取り上げたプログラム〉 プログラム3-1

```
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1; ………………時の加算を計算する
    minute = 45 + 25; ………………分の加算を計算する

    if (minute >= 60) { ………………分の結果が 60 以上ならば
        hour++; ………………時を 1 時間繰り上げ
        minute -= 60; ………………分を 60 分戻す
    }

    printf("%d:%d\n",hour,minute);………………計算結果を表示する
}
```

〈本章で取り上げたプログラム〉 プログラム3-2

```
#include <stdio.h>

main()
{
    int hour,minute; ………………時刻を入れる変数
    int tape_len,tape_num; ………………テープの数を入れる変数
              ………………テープの長さを入れる変数

    hour   = 13; }
    minute = 30; } ………………スタート時刻を 13:30 にする

    tape_len = 46/2; ………………テープの長さは片面ごとなので 46÷2
    tape_num = 3*2; ………………テープの数×2 が交換する回数
```

```
    while (tape_num >0 ) { ……………… テープの数が正の間繰り返す
        tape_num--; ……………… テープの数を1つ減らす

        printf("%02d:%02d にテープを交換してください\n",hour,minute);
                                          ……………… 時刻を表示する
        minute = minute + tape_len; ……………… テープの長さを加算する
        if (minute >= 60) { ……………… 分の結果が60以上ならば
            hour++; ……………… 時を1時間繰り上げ
            minute -= 60; ……………… 分を60分戻す
        }
    }
}
```

〈本章で取り上げたプログラム〉　プログラム3-3

```
#include <stdio.h>

main()
{
    char c; ……………… 入力した文字を入れる変数
    int hour, minute; ……………… 時刻を入れる変数
    int tape_len,tape_num; ……………… テープの数を入れる変数
                ……………… テープの長さを入れる変数
    hour=13;
    minute=30; } ……………… スタート時刻を13:30にする

    tape_len = 46/2; ……………… テープの長さは片面ごとなので46÷2
    tape_num = 3*2; ……………… テープの数×2が交換する回数

    while (tape_num >0) { ……………… テープの数が正の間くり返す
        tape_num--; ……………… テープの数を1つ減らす

        do {
            c = getchar(); ……………… キーボードから1文字入力する
            if (c=='?') ……………… 文字が「?」だったら時刻を表示
                printf("%2d:%02d",hour,minute);
            else ……………… 「?」でなければ
                putchar(c); ……………… 文字をそのまま表示
        } while (c!='\n'); ……………… 改行キーが押されるまでくり返す

        minute = minute + tape_len;
        if (minute >= 60) {
            hour++;                      } ……………… 前のプログラムと同じ
            minute -= 60;                  時間の加算を行なう
        }
    }
}
```

第4章

C言語マスター編

“

　C言語を征服する登山も中腹にさしかかりました。前章までの解説を足がかりに、一歩一歩着実に登ってください。

　本章では、C言語の最も重要で基本的な部分を、詳しく解説します。本章を読み終える頃には、プログラム作成に必要な機能をほとんどマスターしてしまうことになります。C言語処理系を利用できる方は、自分のアイディアでどんどんプログラムを作ってみるとよいでしょう。

　なお、解説の都合上、文法的な解説をせずに新しい構文を導入する部分がありますが、本章の後の方であらためて詳しく解説するので、だいたいの役割を理解して先へ進むようにしてください。

”

本章で解説する項目

<table>
<tr><th></th><th>データ型</th><th>部品化機能</th><th>制御構造</th></tr>
<tr><td rowspan="3">4.1</td><td rowspan="2"></td><td>引数と戻り値
引数を持つ関数の定義
return</td><td></td></tr>
<tr><td colspan="2">関数呼び出し</td></tr>
<tr><td colspan="2">関数と変数の関係
関数と引数の関係
グローバル変数とローカル変数</td><td></td></tr>
<tr><td>4.2</td><td>配列
配列宣言
配列要素
配列の初期化
文字列</td><td></td><td></td></tr>
<tr><td>4.3</td><td></td><td></td><td>whileとdo～while
for
break
return
if
switch～case
条件式
論理演算子</td></tr>
</table>

4.1

関　　数

C言語の3つの機能のなかでも、関数の役割はとくに重要です。そこで、まずは関数から徹底的に解説していきます。

4.1.1　関数による計算処理のカプセル化

引数と戻り値

3章で、プログラムを関数単位に分割することによってプログラム部品として再利用する仕組みを解説しました。関数呼び出しには、もうひとつの役割があります。時間の加算を行うaddclock()関数を作成しながら、この役割を解説しましょう。

図4-1は、addclock()関数を呼び出して値をやりとりする様子を示しています。

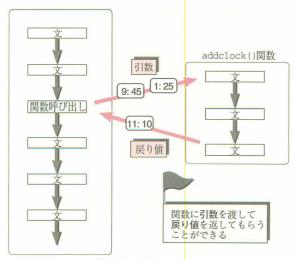



図4-1　引数と戻り値

図のように、関数を呼び出す際には**引数**（ひきすう）としていくつかの値を渡すことができます。関数は、受け取った値を元に計算を行います。そして、計算した結果を、**戻り値**として関数を呼び出したプログラムに返すことができます。

図 4-1 では、addclock()関数に 2 つの時間を引数として渡し、それを加算した結果を戻り値として返しています。

関数の役割

関数という言葉は数学でいう関数からきていますが、C言語では図 4-2 のようなものを考えるとよいでしょう。関数は、値を入力すると対応する値を出力として返してくれる魔法の箱です。

図 4-2　関数の役割

一度関数を作成してしまえば、関数内のプログラムの詳細は忘れてしまうことができます。関数としてカプセル化することにより、他の処理に注意を集中できるのです。

引数を持つ関数の定義

[書式] 関数名([型名 変数名 [, 型名 変数名 ...]]) { 関数本体 }

引数を持つ関数を定義するための書式を図 4-3 に示します。引数は、関数名の後ろの「( )」の中に、変数宣言と似た形式で宣言します。変数宣言と違うのは、各引数を「,」（カンマ）で区切ることと、1 個 1 個それぞれの引数に

型を指定しなければならないことです。通常の変数のように、同じ型の変数をまとめて宣言することはできません。

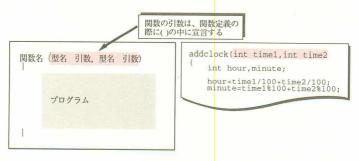



図 4-3　関数定義

return

［書式］return 式;

関数の中で計算した値を呼びだしたプログラムへ返すには、return（リターン）文を使います。return 文で指定した値が「関数の戻り値」として呼びだしたプログラムに返される様子を、図 4-4 に示しました。

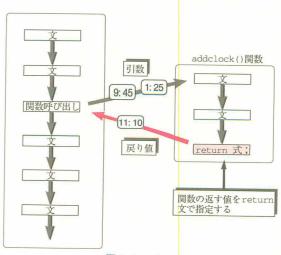



図 4-4　return

関数の型

[書式] 型名 関数名([型名 変数 [, 型名 変数 ...]]) { 関数本体 }

戻り値を返す関数を定義する場合は、戻り値の型を宣言しておかなければなりません。戻り値の型は下の図 4-5 のように、関数定義の際に「関数の型」として宣言します。

図 4-5　関数の型

なお、これまでの関数定義（87 ページ図 4-3 参照）の書式では、関数の型は指定していませんでした。このように、関数の型は省略することができ、省略すると int 型であると解釈されます。

4.1.2 関数によるプログラム部品化の実際

引数を渡して戻り値を返すという関数の仕組みがわかれば、addclock()関数を作成する準備は完了です。しかし、ここで問題がひとつあります。関数は戻り値をひとつしか返せないのです。時間の計算に必要な時と分の両方を、関数の戻り値としてひとつずつ返すことはできません。

そこで図4-6では、時と分をひとつの数値で表す方法を使います。1時間は60分ですから、時に60を掛けることによって分に換算することもできますが、ここでは時に100を掛けるという方法（以後100倍法と呼びます）を採用します。こうすると、9時45分を10進数で945と表すことができるからです。

```
int addclock(int time1, int time2)
{
    int hour, minute;

    hour=time1/100+time2/100;
    minute=time1%100+time2%100;

    if(minute >= 60){
        hour++;
        minute-=60;
    }
    return hour*100+minute;
}
```


図 4-6　addclock()関数

関数を使った例題プログラム

本項の例題プログラムは、次ページの図4-7のリストを加えて完成します。tape()関数は、3章の3.2節で作成したテープの交換時刻を計算するプログラムをaddclock()関数を使うように変更し、さらに関数として再利用できるようにしたものです。また、prtime()関数は、文字「?」を時刻に置き換えて表示するプログラムを関数として部品化してあります。

```
#include <stdio.h>

main( )
{
    int t;
    int tape_num;

    t = 1330;
    tape_num =2;

    tape(t,46,tape_num);
    tape(t,60,tape_num);
}

tape(int start_time,int tape_len,int tape_num)
{
    int t;

    print("[%d分テープの場合]\n",tape_len);


    t = start_time;
    tape_len/=2;
    tape_num*=2;

    while (tape_num>0) {
        tape_num--;
        prtime(t);
        t =addclock(t,tape_len);
    }
}

pritime(int t)
{
    char c;

    do {
        c=getchar( );
        if (c=='?')
            printf("%2d:%02d",t/100,t%100);
        else
            putchar(c);
    } while (c!='\n');
}
```

＜実行結果＞

```
A>turn2 > turn2.out
鈴木くんは、?からです。
南くんは、?からです。
智恵子ちゃんは、?からです。
山田くんは、?からです。
鈴木くんは、?からです。
南くんは、?からです。
智恵子ちゃんは、?からです。
山田くんは、?からです。
A>type turn2.out
 [46分テープの場合]
鈴木くんは、13：30からです。
南くんは、13：53からです。
智恵子ちゃんは、14：16からです。
山田くんは、14：39からです。
 [60分テープの場合]
鈴木くんは、13：30からです。
南くんは、14：00からです。
智恵子ちゃんは、14：30からです。
山田くんは、15：00からです。
A>
```

キーボードからの入力

プログラムの出力

図 4-7　tape()関数、prtime()関数と実行例

関数呼び出し

［書式］関数名(式　[, 式 ...])

図 4-8 は、例題プログラムの中で addclock()関数を呼び出している部分です。図のように、関数を呼び出して返ってきた戻り値を、変数の値を利用するのと同じように利用することができます。

```
while(tape_num>0){
      tape_num--;
      prtime(t);
      t=addclock(t,tape_len);
  }
```

図 4-8　関数呼び出し(1)

ところで、printf()関数も、ライブラリ関数としてあらかじめ用意されているプログラム部品です。addclock()関数はあたかも変数のように利用しているのに対して、図 4-9 のように printf()関数は C 言語の命令であるかのように呼び出しています[*1]。戻り値を返さない関数や、戻り値を返しても、それを使わない場合には、式の中ではなく単独に関数を呼び出すことができるのです。ただし、戻り値を返す関数をこのような方法で呼び出すと、戻り値は捨てられてしまいます。

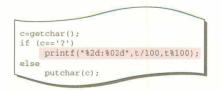



図 4-9　関数呼び出し(2)

関数と変数の関係

関数と変数には、密接な関係があります。関数と変数の関係を図に表すと、

＊1　printf()関数は戻り値を返しますが、その値を使うことはめったにありません。

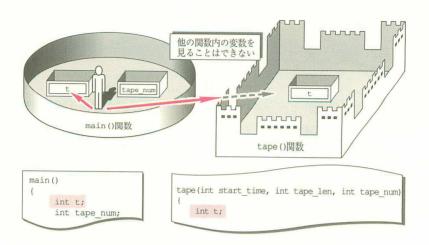



図 4-10　関数と変数の関係

図 4-10 のようになるでしょう。

変数は関数という城壁に囲まれています。つまり、関数の中では変数が見えていますが、関数の外側からは壁にさえぎられて見えません。他の関数では、この変数の値を取り出したり代入したりすることはできないのです。変数は、関数の中だけで有効です。

逆の見方をすれば、そのおかげで、関数のプログラム部品としての独立性が確保されます。つまり、他の関数によって変数の値がいつのまにか変更されるといったことがなく、関数のプログラム部分だけで処理が完結しているのです。

例題のプログラムでは、main()関数で変数 t を宣言していますが、tape()関数でも同じ名前の変数 t を宣言しています。変数は関数内だけで有効ですから、同じ名前であっても両者はまったく無関係で、それぞれ独立に存在しています。一方の値を変更しても、他方には影響を与えません。

関数と引数の関係

関数の引数も、関数の中だけで有効な一種の変数で、式の中で使ったり、値を代入したりすることができます。この意味から、引数のことを**引数変数**と呼ぶことがあります。引数変数と通常の変数の違いは、関数が呼び出され

た時点ですでに値が設定されているかどうかだけです。

関数が呼び出された時点では、変数の値は決まっていません。変数には値を代入しない限り、どんな値が入っているかわからないのです。同じ関数が2度目に呼び出されたときも、以前変数に代入した値は残っていません。これに対して、引数変数には関数呼び出しの引数として指定された値が設定された状態で関数が呼び出されます。

次の図4-11は、tape()関数が呼び出され、実行を開始した時点の様子を表しています。引数変数であるstart_timeなどには、main()関数の中で指定された値が設定されていますが、変数tには値が設定されておらず、でたらめな値が入っています。

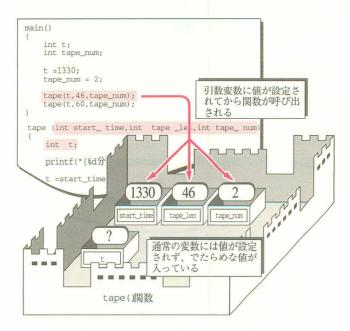
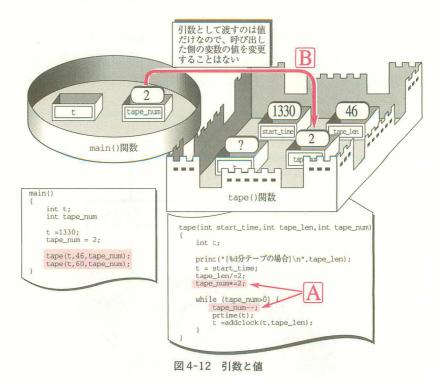



図4-11　関数呼び出しと引数の受け渡し

引数変数は変数の一種ですから、関数の中だけで有効です。ところが、次のような場合には勘違いしやすいので注意してください。

次の図4-12のように、main()関数からtape()関数を呼び出す際に、引数と

して変数 tape_num の値を渡しており、tape()関数は引数変数 tape_num をプログラムの中で変更しています（図の A 部分）。しかし、90 ページの図 4-7 を見てわかるように、main()関数の変数 tape_num は変更されていません（もし変更されているとすると、2 回目の tape()関数の呼び出しでは、tape_num が 0 になっているはずです）。

図 4-12　引数と値

このプログラムでは、変数そのものを引数として渡しているようにも思えますが、main()関数の変数 tape_num と tape()関数の引数変数 tape_num はまったく関係のない別の変数です。実際には、main()関数の変数 tape_num の値を tape()関数の引数変数 tape_num に代入してから（図の B 部分）、tape()関数を呼び出しています。

あたりまえのことのように思えるかもしれませんが、C 言語をはじめたばかりのころには、引数に変数を渡すと、関数の中でその変数の値を変更できるように思ってしまうことがあるので注意してください。

グローバル変数とローカル変数

変数は関数内だけで有効で、関数の中でしか見えないと述べましたが、実はどの関数からも見える変数を宣言することができます。

関数の中で宣言した変数は、関数の中だけで有効な**ローカル変数**と呼ばれます。これに対し、関数の外で変数を宣言すると、どの関数でも有効な**グローバル変数**となります（図 4-13）。

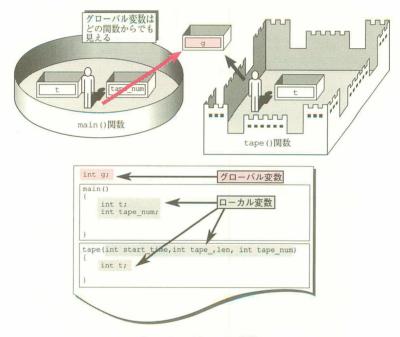
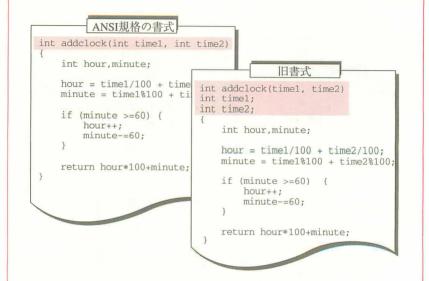



図 4-13　グローバル変数

グローバル変数はローカル変数と違って、関数の実行を終了しても値が消えることはありません。また、どの関数からでも値を参照したり、変更したりすることができます。このことから、グローバル変数は、プログラム全体で共通に利用する変数として使います。しかし、グローバル変数を使うと、当然関数のプログラム部品としての独立性は低くなります。変数の値がどこで変更されているかを把握しづらくなるからです。

ですから、関数間で情報を受け渡すには、なるべく引数だけで受け渡すよ

うにして、どうしても引数だけではうまくいかない場合に限って、グローバル変数を使うようにしましょう。

関数定義の旧書式

関数定義の書式には、これまで解説した書式以外に図に示すような書式があります。この書式はANSI以前のC言語で使われていたもので、現在でもまだ広く使われています。本書ではこの書式を旧書式と呼ぶことにします。

旧書式と現在の書式の違いは、引数の宣言方法です。引数の宣言方法が変更されたのは、7章で解説するプロトタイプ宣言と同じ形にするためです。詳しくは7章で解説します。もし、みなさんの使っている処理系がANSI準拠のものでない場合は、図右のような書式を使ってください。なお、現書式に対応している処理系でも、互換性保持のため、旧書式を扱うことができます。しかし、これからプログラムを書くならば、なるべくANSI準拠の書式を使うことをお勧めします。

4.2

配　　列

基本データ型から一歩進んで、複合データ型を解説しましょう。複合データ型とは、いくつかの情報をまとめて1つの変数として扱うためのデータ型です。

複合型の変数には、配列や構造体などがありますが、構造体は6章で解説することにして、本章では配列を解説します。また、配列の一種である、文字列についても本章で解説します。

4.2.1　配列

配列とは

配列の例題として、時刻表をプログラムで表現してみましょう。

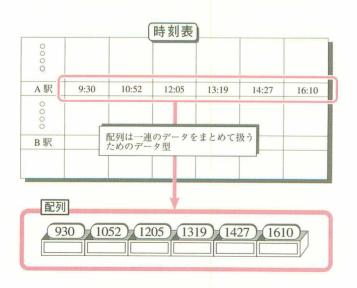



図4-14　配列

時刻表は、電車やバスが発車する時刻という情報をたくさん並べたものです。多くの時刻情報が載っていますが、その情報ひとつひとつに変数を用意するのはめんどうです。このような情報を処理するには、いくつもの連続した情報をまとめて扱う方法が必要です。

図4-14は、ある路線の電車の時刻表です(わかりやすくするために電車の本数をうんと少なくしました)。A駅を電車が発車する時刻が図のように決められているとします。このような一連のデータをまとめて格納できるように、変数をいくつも並べたデータ型を**配列**と呼びます。

本節では、時刻表を検索するプログラムを作成しながら、解説を進めます。

配列宣言

[書式] 型名 変数名[要素数] ;

それではさっそく配列を使うための文法を解説しましょう。配列は同じ型の変数をいくつも並べたもので、並べた変数全部にまとめて1つの名前を付けます。

配列の宣言の書式は、これまでの変数宣言とほとんど同じで、名前の後に「[]」で囲んで「配列の大きさ」を指定します。配列の大きさとは、まとめて扱う情報の数のことです。変数を宣言すると情報の入れ物が1つだけ用意されますが、配列を宣言すると図4-15のように指定した数の変数を一度に用意することになります。

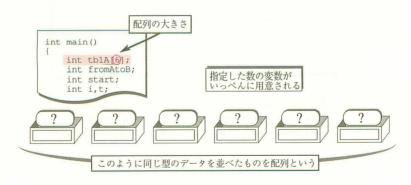



図4-15　配列

例題プログラムでは、A 駅の時刻表を格納するために、int 型の変数 6 個からなる配列 tblA を宣言しています。

配列要素

［書式］変数名［添字］；

配列中のひとつひとつの変数のことを**配列要素**といいます。各配列要素は、**配列名**と『先頭から何番目』という番号を使って図 4-16 のような書式で表します。

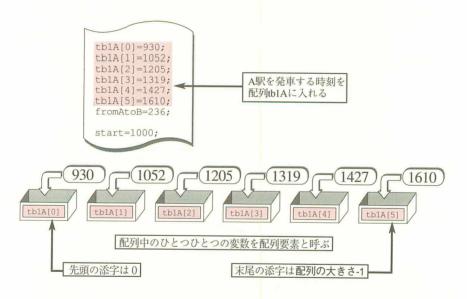



図 4-16　配列要素

配列要素の番号のことを、配列の**添字**といいます。配列の添字は、かならず 0 からはじまることをよく覚えておいてください。配列の末尾の要素の添字は「配列の大きさ－1」となります。最初のうちはこのことを忘れやすいので注意が必要です。

章末のプログラム 4-2 では、配列 tblA の各要素に B 駅行きの電車が A 駅を発車する時刻を格納しています。時刻は前章で使った 100 倍法を使って表

しています。配列 tblA の大きさは 6 ですから、0 から 5 までの添字で要素を指定することになります。1 から 6 ではない点に注意してください。

時刻表検索プログラム

例題プログラムは、最寄り駅である A 駅を出発する時間を与えると、A 駅発の次の電車の発車時刻と、B 駅に着く時刻を求めるというものです。

図 4-17 は、時刻表を検索している部分です。この図は、典型的な配列処理方法を表しています。配列全体を処理するには、図のように繰り返し処理を使って、先頭の要素から順番にひとつずつ処理していきます。

配列要素を先頭から順番に指定するために、変数 i を添字に使うことに注目してください。変数 i の値を 0 から 1 つずつ増やしていくことで、配列要素を順番に指定することができます。

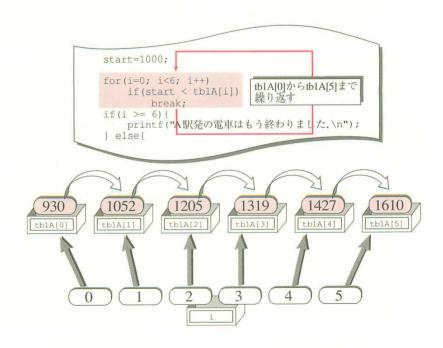



図 4-17　配列の処理

繰り返し処理のためにfor文を使っていますが、詳しい書式については4.3節で解説します。図4-17のように、繰り返すたびに変数iの値を1つずつ増やす処理を行っていると理解してください。

配列の中から、A駅を出発する時刻（変数start）よりも遅い時刻の要素を探すわけですが、100倍法で表した時刻は、そのまま大小を比べることによって時刻の前後を判断することができます。したがって、図のようにif文による条件判断で、出発時刻よりも遅い時刻かどうかを判断しています。

めざす要素が見つかったら、それ以上繰り返す必要はありません。そこで、繰り返し処理を中断するために、break文を使っています。詳しくはやはり4.3節で解説しますが、break文を実行すると、繰り返しの途中であっても、その時点で繰り返し処理全体を中断して次の処理へ進みます。

図4-18は繰り返しを終了、または中断した後の処理の部分です。めざす要素が見つかった場合は、途中で中断するので、変数iはその要素の添字になっています。めざす要素が見つからなかった場合、つまり途中で中断せずに最後まで繰り返して終了した場合には、変数iは末尾の要素の添字よりも大き

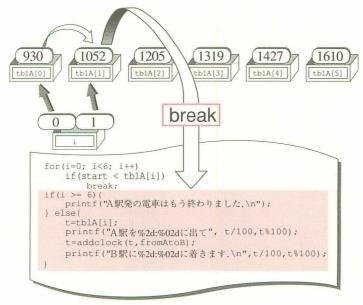
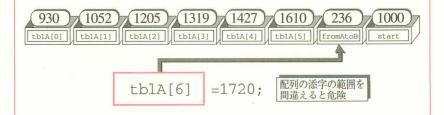



図4-18　繰り返し後の処理

くなっています。したがって、変数iが末尾の要素よりも大きいかどうかを比べることによって、繰り返しの最中に目指す要素が見つかったかどうかを調べることができます。

配列利用上の注意

C言語は、配列の添字が範囲内に収まっているかどうかをチェックしません。配列を使った処理を行う場合、このことに十分注意してください。

図のように、配列の末尾の要素よりも大きな添字を誤って使ってしまっても、文法エラーとはなりません。このようなプログラムを書いてしまうと、配列として確保されている領域をはみ出して他の変数などを壊してしまうこともあります。

例題のプログラムのように、変数を添字として配列を処理するプログラムでは、変数を変化させる範囲を間違えると、とても危険です。

BASICなどのプログラミング言語では、配列要素を参照するたびに添字が範囲をはみ出していないかどうかをチェックしてくれます。したがって、誤って他の変数を壊してしまうようなことはありません。しかし、その代償として、プログラムの実行時間が余分にかかります。

C言語では、高速な実行速度を提供する代わりに、配列要素の添字のチェックをプログラマの責任に任せています。注意深くプログラムを作成するように心がけてください。

配列の初期化

［書式］型名 変数名［要素数］ = {初期値, 初期値, ...}；

プログラム 4-2 では、配列に時刻表を格納するために、処理部で配列要素に数値をひとつひとつ代入しています。この方法では、ほとんど同じ文を何度も書かなければならないので面倒です。そこで、配列に情報を格納するもうひとつの方法を紹介しましょう。

図 4-19 を見てください。このように、配列の宣言時に、1 つの式でいくつもの情報をまとめて格納することができます。この方法は、配列の宣言と同時に配列要素の初期値を設定することになるので、**配列の初期化**と呼びます。

図のように配列宣言時に初期化を行う場合、[]の中に書く配列の大きさを省略することができます。この場合、初期データとして並べたデータの数が自動的に配列の大きさとなります。

配列の初期化を安易に使うと、効率のよくないプログラムになってしまいます。なぜかというと、ローカル変数は関数が呼び出された時点で有効になるので、関数を呼び出すたびに配列への代入が実行されるからです。このた

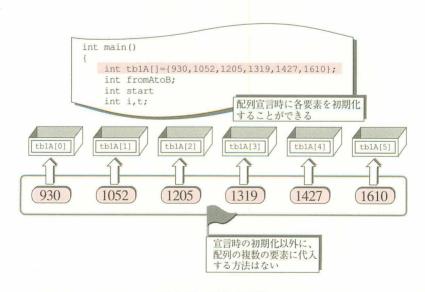



図 4-19　配列の初期化

め、ANSI以前の処理系では、ローカル変数の配列を初期化できないようになっていました。

そこで、初期値を持つ配列は図4-20のようにグローバル変数にします*2。グローバル変数なら関数の実行を終了しても値が保存されるので、関数を呼び出すたびに代入されることはありません。

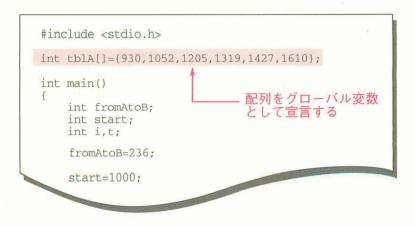



図4-20　初期値を持つ配列の宣言

完成した時刻表検索プログラムを図4-21に、また、時刻表検索プログラムの実行例を図4-22に示します。なお、このプログラムで呼び出しているaddclock()関数は、89ページの図4-6と同じものなので省略します。

＊2　グローバル変数の初期化処理、すなわち変数への値の代入は、コンパイル時に実行されます。また、これ以外に、static型変数にする方法もあります。詳しくはAppendix 8に挙げた解説書を参考にしてください。

```
#include <stdio.h>

int tblA[]={930,1052,1205,1319,1427,1610};

int main()
{
    int fromAtoB;
    int start;
    int i,t;

    fromAtoB=236;

    start=1000;

    for(i=0; i<6; i++)
        if(start < tblA[i])
            break;
    if(i >= 6){
        printf("A駅発の電車はもう終わりました.\n");
    } else{
        t=tblA[i];
        printf("A駅を%2d:%02dに出て", t/100,t%100);
        t=addclock(t,fromAtoB);
        printf("B駅に%2d:%02dに着きます.\n",t/100,t%100);
    }
}
```

A駅からB駅まで電車でかかる時間を入れる変数
出発時刻を入れる変数
A駅からB駅まで2時間36分かかる
10:00に出発する
iを0から5まで繰り返す
出発時刻よりあとの電車があったら、繰り返しを中断
iが6に達していたらもう電車はない
次の電車の発車時刻
B駅に着く時刻を計算する

図 4-21　時刻表検索プログラム

```
A駅を10:52に出てB駅に13:28に着きます.
```

図 4-22　時刻表検索プログラムの実行例

4.2.2　文字列

文字列とは

char型の配列、すなわち文字の配列は連続した文字の処理に適しており、非常によく使われます。文字の配列の中でも、次ページの図4-23の形式を満たしているものを**文字列**と呼びます。

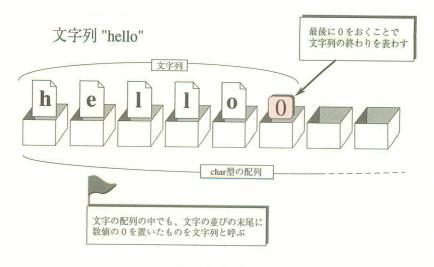



図4-23　文字列

図のように、文字の並びの末尾に**数値の0**を置いたものがC言語の文字列です。文字を入れる変数に数値を入れるのはへんだと思うかもしれませんが、ここでは特殊な例だと思っておいてください。

文字の並びの末尾に数値の0を入れるのは、文字列の終わりを示すマークにするためです。C言語は文字列や配列の長さの管理を行わないので、マークが必要なのです。もしも、マークを入れない場合は、文字列の長さを別途管理しておかなければなりません。

たとえば、他の関数に引数として文字列を渡す場合を考えてみてください。文字列を受け取った関数は、文字列の終わりを知る手段が必要です。末尾にマークを入れておけば、マークを探すことによって文字列の長さがわかります。ところが、もしマークを入れないとすると、文字列の長さをもうひとつの引数として渡さなければならないでしょう。引数をいくつも渡すのは面倒ですし、プログラムミスを発生させる可能性も高いので、マークを入れる方式を採っているのです。

文字列の書式

char型の配列に文字列を代入するには、図4-24のようにします。

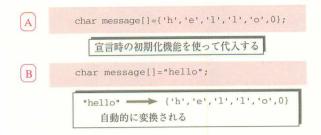



図 4-24　文字型配列の初期化

前節で解説したように、配列の複数の要素をまとめて代入できるのは、宣言と同時に初期化するときだけです。図 4-24 の A のように、配列の各要素である文字をひとつずつ並べて初期化することができます。

図 4-24 の B は、文字列を表す記法を利用する方法で、これまでにも使っていた「"」（ダブルクォーテーションマーク）で文字列を囲む方法です。「"」で囲んだ文字列は、末尾に数値の 0 を置く C 言語の文字列形式に自動的に変換されます。

テープ交換時刻表示プログラム

［書式］ char 変数名［要素数］；

この節では、文字列を使った例題プログラムを作成します。文字列すなわち char 型の配列宣言は、上の書式のように int 型の配列宣言となんら変わるところはありません。

3 章で作成したカセットテープ交換と書記交代の時刻表示プログラムを、文字列を使ったものに改造してみましょう。次ページの図 4-25 の prtime_s() 関数は、文字'?'を時刻に置き換えて表示する関数を、キーボードから直接 1 文字ずつ受け取るのではなく、引数として文字列を受け取るように変更したものです。

この関数でも、for 文を使って繰り返し処理を行っています。for 文については次の 4.3 節で詳しく解説しますので、ここでは図 4-26 のように文字列中のひとつひとつの文字の数だけ処理を繰り返すことを理解してください。

```
prtime_s(char str[],int t)
{
    char c;
    int i;

    for(i=0; str[i]!=0; i++){
        c=str[i];
        if(c=='?')
            printf("%2d:%02d",t/100,t%100);
        else
            putchar(c);
    }
}
```

図 4-25　prtime_s()関数

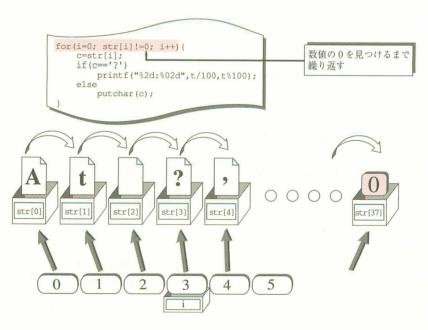



図 4-26　文字列の処理

文字列定数

文字列を、関数への引数として渡す方法を解説しましょう。90ページの図4-7のtape()関数を、prtime_s()関数を使うように改造したtape_s()関数を2つの方法で作成してみます。

まず、考えられる方法は、次の図のように配列型変数を用意して、それを引数として渡す方法です。

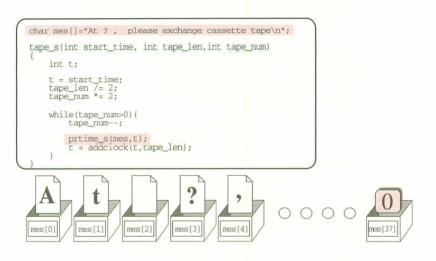



図4-27　引数と配列

もうひとつの方法として、次の図4-28のように 引数に"..."で囲んだ文字列を直接指定する方法があります。これまでの例題プログラムでも、printf()関数を呼び出す際にこの方法で文字列を引数として渡していました。

このような"..."で囲んだ文字列のことを**文字列定数**と呼びます。プログラム中で文字列定数を使用すると、図のように文字列が格納された配列型変数が自動的に用意されます。わざわざ配列型変数を宣言して用意しておかなくても、文字列定数を使えば簡単に文字列を渡すことができるのです。

文字列定数は、文法的には配列名と同じ扱いと考えることができ、配列名を使う場面ではどこでも文字列定数を使うことができます。名前のないchar型配列変数のようなものだと思えばよいでしょう。

```
tape_s(int start_time, int tape_len,int tape_num)
{
    int t;

    t = start_time;
    tape_len /= 2;
    tape_num *= 2;

    while(tape_num>0){
        tape_num--;

        prtime_s("At ? , please exchange cassette tape\n",t);
        t = addclock(t,tape_len);
    }
}
```

図 4-28　文字列定数

最後にプログラムの残りの部分とその実行例を図 4-29 に示します。

```
#include <stdio.h>

main()
{
    int t;
    int tape_num;

    t=1330;
    tape_num=3;

    tape_s(t,46,tape_num);
}
```

<実行結果>

```
At 13:30 , please exchange cassette tape
At 13:53 , please exchange cassette tape
At 14:16 , please exchange cassette tape
At 14:39 , please exchange cassette tape
At 15:02 , please exchange cassette tape
At 15:25 , please exchange cassette tape
```

図 4-29　カセットテープ交換時刻表示プログラム

制御構造

プログラムの流れをコントロールする構文、制御構造についてまとめて解説します。

これまでの解説では、関数の仕組みや配列の扱い方が中心で、例題のプログラムに登場した新しい構文については詳しく解説しませんでした。これらの構文は簡単なものばかりですから、おおよその働きはつかめていると思います。本節では、こうした制御構造を再びピックアップして解説します。

本節ではじめて登場する新しい構文についても、新しい例題プログラムを作成しながら解説していきます。本節の例題プログラムは「日付の計算」を行うプログラムです。たとえば、3月28日の100日後は何月何日でしょうか？　9月3日は1月18日の何日後でしょうか？　こうした計算は暗算や電卓ではなかなか面倒なので、プログラムにしておくと便利です。

4.3.1　繰り返しの構文

繰り返しの構文は、図4-30のように3種類あります。これらの構文を徹底的に解説しましょう。

while	do～while
while(条件式) 　　　文または ブロック	do 　　　文または ブロック while(条件式);
for	
for(初期設定;繰り返し条件;次の繰り返しの準備) 　　　文または ブロック	

図4-30　繰り返しの構文

while と do〜while

while 文と do〜while 文は、どちらも条件が満たされている間繰り返すという構文ですが、両者の違いは次の図 4-31 のように条件のチェックをするタイミングにあります。「while 文」ではループをまわる前に条件をチェックし、「do〜while 文」ではループを回った後にチェックします。

繰り返し処理

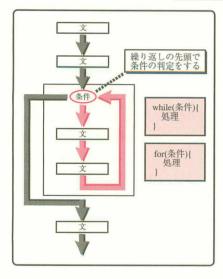
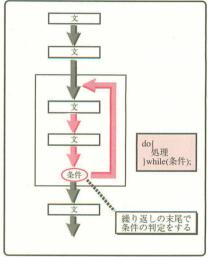



図 4-31　while 文と do〜while 文

これまでの例題プログラムでも図 4-32 のように、これらの構文を使っています。while 文と do〜while 文の違いを確認してください。

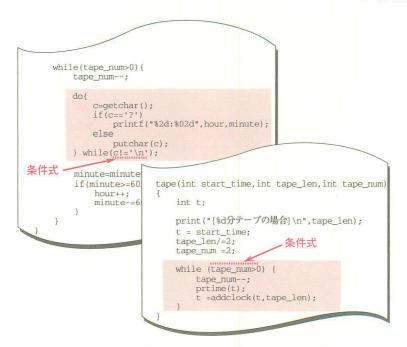



図 4-32　while 文と do～while 文を使った例題プログラム

for

for 文はこれまでの例題プログラムでも使っていましたが、新たに例題プログラムを作成します。

日付の計算を簡単にするために、まず 1 月 1 日からの日数を計算することにします。こうすれば、足し算や引き算を使って日付の計算ができるようになります。この計算を行う ydays()関数を図 4-33 に示します。

プログラムで呼び出している中の mdays()関数は、引数に指定した月の日数を戻り値として返す関数です。閏年かどうかで 2 月の日数が変わるので、年も引数として渡しています。この mdays()関数は後で作成します。

ydays(年，月，日)	1月1日からの総日数を返す

```
int ydays(int year,int month,int date)
{
    int days,i;

    days=0;
    for(i=1; i<month; i++)          } 1月から前月までの日数を合計する *3
        days+=mdays(year,i);        }
    days+=date; <────────── 今月の日数を加える
    return days;
}
```

図 4-33　ydays()関数

ydays()関数では、次の図 4-34 のような計算を行っています。前月までの日数を求めるために、繰り返しの構文である for 文を使って各月の日数を合計していることに注目してください。

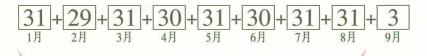
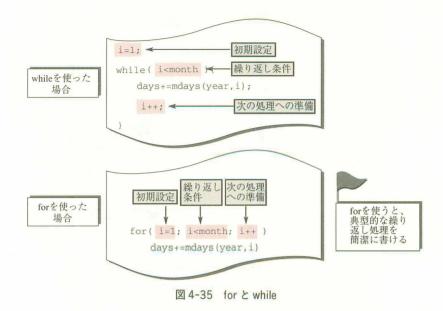
2000年9月3日の場合

for文による繰り返しで計算する

図 4-34　ydays()関数の計算方法

for 文は、図 4-35 に示すように while 文による繰り返しにいくつかの手順を加えた処理を行います。繰り返し処理には配列の要素をひとつずつチェックするような型にはまった処理が多く、そうした処理を簡潔に記述できるように特別に for 文が用意されているのです。

*3　＋＝演算子については 59 ページを参照してください。

図 4-35 for と while

ydays()関数では、変数 i を 1 から month−1 まで変化させて、各月の日数を足しています。このような処理を繰り返しの構文を使って実行するには、すくなくとも**初期設定、繰り返し条件、次の繰り返しへの準備**という 3 つの式が必要です。

まず、i に 1 を代入し、1 月を表すようにします（初期設定）。次に、i が month−1 になるまで繰り返したかどうか、つまり繰り返し処理を続けるかどうかを判定する式（繰り返し条件）が必要です。そして、繰り返しのたびに i が次の月を表すように、i をインクリメントしなければなりません（次の繰り返しへの準備）。

for 文は、繰り返しのかたちを決めるこの 3 つの式をまとめて書くことによって、典型的な繰り返し処理を簡潔に書けるようにしたものです。

for 文による繰り返し

for 文による繰り返し処理がどのような手順で行われるかを、詳しく解説しましょう。ydays()関数を例に、for 文の 3 つの式と繰り返される処理の実行順序を図 4-36 に示します。

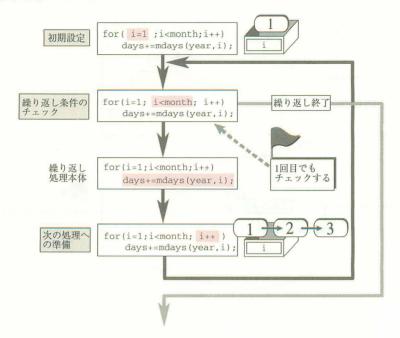



図 4-36　for 文による繰り返し

この図で重要なことは、処理本体を一度も実行しないうちに、まずはじめに繰り返し条件をチェックすることです。たとえ一度目であっても条件を満たさなければ、for 文の処理はそこで終了します。

もう一度前節の時刻表検索プログラムや文字列処理プログラムに戻って、配列の処理をどのような手順で行っているかを確かめてみてください。

for 文の応用

他のプログラミング言語にも、C 言語の for 文と同じような構文が用意されています。その多くは、初期値と終値を指定して、変数を初期値から順にひとつずつ増やしていき、終値になるまで繰り返すというものです。C 言語の for 文も同じような考えから生まれているのですが、繰り返しのかたちを決める 3 つの式がそれぞれ自由に指定できるようになっているため、非常に柔軟な書き方が可能です。

たとえば、変数をひとつずつ増やすのとは逆に、ひとつずつ減らしながら

繰り返すこともできます。この場合には3つの式を次ページの図4-37の(2)のように反対の意味にすればよいのです。さらに、(3)のように変数を2ずつ増やすことも、(4)のように1/2にしながら繰り返すこともできます。

また、(5)のように繰り返しの終了条件を『配列中の特定の値（図の場合は0)が見つかるまで』とすることもできます。図のように、繰り返し条件を『その値と等しくない間』とすると、その値が見つかったところで繰り返しを終了することになります。前節の文字列の処理では、この方法を使っていました。

for文で指定する3つの式は、必要でないものは省略することができます。3つとも省略すると、図の(6)のように無限ループになります。

このように、C言語のfor文は、柔軟な書き方ができることがわかったと思います。このほかにも、いろいろな繰り返しのかたちをfor文を使って書くことができますので、プログラム集などのプログラム例を読んで研究してください。

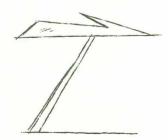
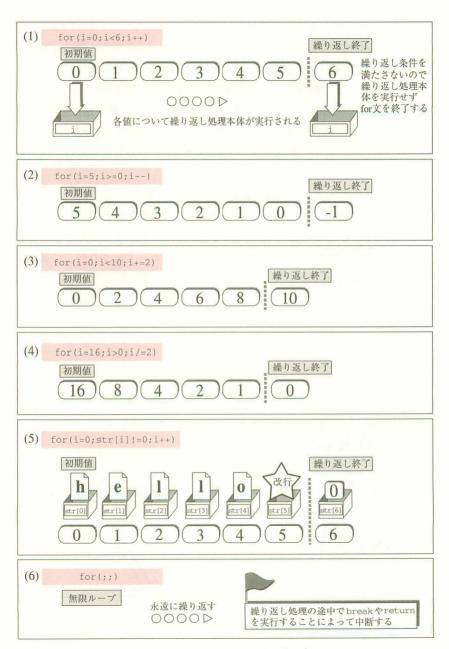



図4-37　いろいろなfor文の使い方

break

for 文について、ひととおり解説したところで、繰り返しを中断する命令であるbreak（ブレーク）を解説しましょう。break文を実行すると図4-38のように繰り返しを中断して、次の処理へと進みます。「while文」、「do～while文」、「for文」のいずれの場合も、break文によって繰り返しを中断します。

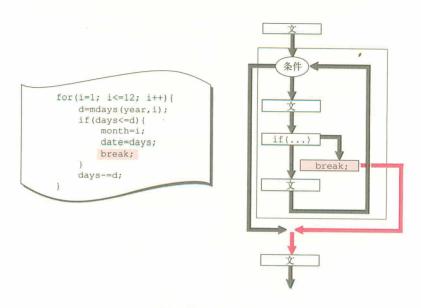



図 4-38　break

次ページの図4-39は、ydays()関数とは逆に1月1日からの日数を元に日付を表示するプログラム、prdate()関数です。このプログラムでは、1月から12月まで順番に各月の日数を加えながら、求める月までくると、break文によって繰り返しを中断します。

図4-40のように、繰り返し処理の内部でさらに繰り返しを行っているような場合を**ネストした繰り返し**といいます。ネストした繰り返しのなかでbreak文が実行された場合、最も内側の繰り返しだけを中断します。

prdate（年，総日数）	1月1日からの総日数に対応する日付を表示する

```
int prdate(int year,int days)
{
    int d,month,date;
    int i;

    for(i=1; i<=12; i++){
        d=mdays(year,i);      ← i月の日付を求める
        if(days<=d){          ← daysがi月の日数より少なければ
            month=i;          } ← i月days日が求める日数
            date=days;        }
            break;
        }
        days-=d;              ← 総日数からi月の日数を引く
    }
    printf("%d年%d月 %d日 ",year,month,date);
}
```

図4-39　prdate()関数

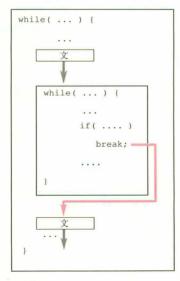



図4-40　ネストした繰り返し

return

例題のプログラムとは関係ありませんが、return 文についてもここで解説しておきましょう。関数の中で return 文を実行すると、たとえ繰り返しの最中であっても、関数の処理を終了し、呼び出した元のプログラム処理に戻ります。

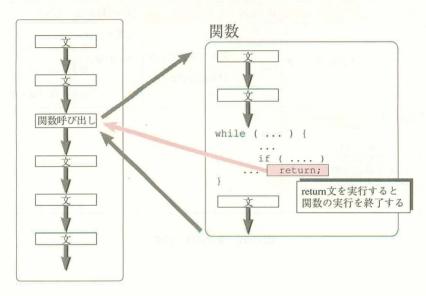



図 4-41　return

値を返さない関数でも、戻り値を指定せずに「return ;」を実行することによって、関数の実行を終了させることができます。これは、繰り返しの途中で処理を続行する必要がなくなったときなどに便利ですが、プログラムがわかりにくくなる可能性もあるので、むやみに関数の途中から return するのはやめたほうがよいでしょう。

4.3.2　条件分岐の構文

条件分岐の構文には図4-42のようにif文とswitch文の2種類があります。本節では、条件分岐の構文を徹底的に解説します。

if	if else	if～else if～else

if(条件式)
　　文またはブロック
	if(条件式)
	　　文またはブロック
	else
	　　文またはブロック
		if(条件式1)
		　　文またはブロック
		else if(条件式2)
		　　文またはブロック
		else
		　　文またはブロック

switch		

switch(条件式){
case 定数式:文の並び;
case 定数式:文の並び;
default:文の並び;
}		

図4-42　条件分岐の構文

if

条件分岐に使う「if～else文」のifは、英語で『もし～ならば』という意味で、elseは『もしくは』という意味です。

図4-43は、if文による条件分岐によって4つの文のうち1つだけが実行されることを表しています。条件式1から順に真偽を判定していき、真となったところで対応する文が実行されるのです。すべての条件式が偽であれば、処理4が実行されます。

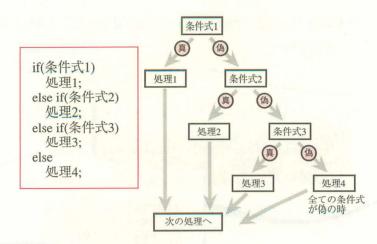



図 4-43　if～else 文（その 1）

「else if」の部分はいくつでも書くことができますし、逆に必要がなければ「else if」や「else」の部分は、次の図 4-44 のように省略することができます。例題のプログラムでは、このうち最も簡単な書式を使っています。

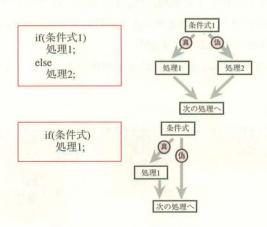



図 4-44　if～else 文（その 2）

次の図4-45は、これまでの例題プログラムでif文を使って条件分岐を行っていた部分です。

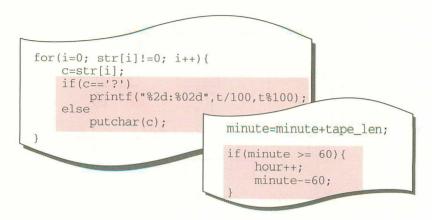



図4-45　if文を使ったプログラム例

if文とブロック

if文の各条件式に対応する処理には、1つの文だけでなくブロック[*4]を書くこともできます。2つ以上の文を条件式と対応させたい場合は、必ずブロックにしなければなりません（図4-46）。

同様にelse ifや最後のelseに対応する処理にもブロックを書くことができます[*4]。

図4-46　if文とブロック

*4 ブロックについて忘れてしまった人は、45ページを見てください。

switch〜case

```
［書式］ switch (式) {
        case 定数式: 文の並び
        case 定数式: 文の並び
        default: 文の並び
        }
```

条件分岐のもう１つの構文は「switch〜case文」です。switch〜case文は、多くの条件を使って処理を分岐させたい場合に便利な構文です。switch〜case文の実行の仕組みを次ページの図4-47に示しました。

switch（スイッチ）は、「スイッチを切り替える」というときのスイッチと同じで、条件によって処理を『切り替える』ことを意味します。case（ケース）は『場合』という意味で、「〜の場合、〜の場合」のように条件ごとに処理を書き並べることを意味しています。

図に示したように、各「case文」のことを**ラベル**と呼びます。ラベルとは名札のことで、名前（式）と一致する名札（値）を見つけて対応する処理を呼び出すところからきています。

caseラベルに指定する値には、定数を書きます。つまり、変数を使った式を書くことはできません。

各caseラベルには、対応する実行文をいくつも書き並べることができます。図では、最後にbreak文が書かれています。このbreak文はcaseラベルに対応する処理を終了して、switch文の処理全体を終わらせる働きをします。繰り返しを中断するbreak文ではないので注意してください。

このbreak文がなければ、図4-48に示すように、後に続く他のcase文に対応するプログラムまで実行されてしまいます。

文

switch(式) {

これをラベルと呼ぶ

case 1:

式の値が1の時

文

文

break;

case 2:

式の値が2の時

文

文

break;

case 3:

式の値が3の時

文

文

break;

default:

それ以外の時

文

文

break;

}

文

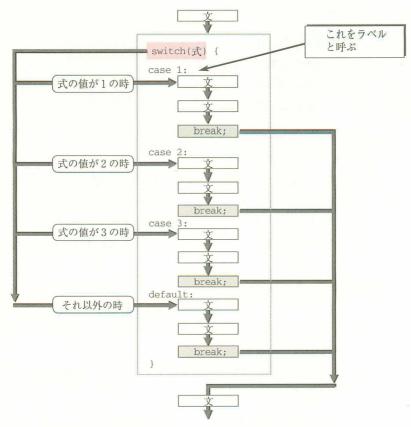
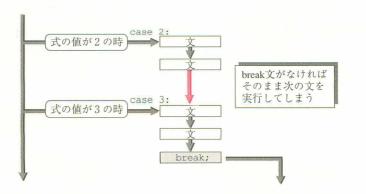



図 4-47　switch〜case

図 4-48　switch〜case 文における break 文（その 1）

誤って他のcaseラベルに対応する処理を実行してしまわないように、switch～case文ではbreak文を忘れないように注意しなければなりません。しかし、この仕組みを逆に利用して、次の図4-49のように複数のcaseラベルで共通の処理を書くこともできます。

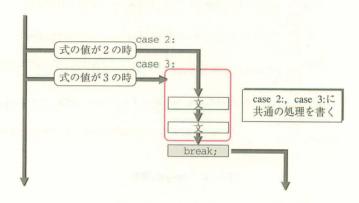



図4-49 switch～case文におけるbreak文（その2）

最後のdefault（デフォルト）ラベルは、どのcaseラベルにも一致しなかった場合に呼び出されます。defaultとは、『省略時の』という意味で、「if～else if～else文」での最後のelseにあたる処理です。caseラベルとして書ききれない部分の処理や、エラー処理を行うのに便利です。

defaultラベルの処理ではbreak文は必要ありませんが、プログラムを改造していくうちにdefaultラベルからcaseラベルに変更することも少なくないので、ミスを防ぐためbreak文を書くようにした方がよいでしょう。

switch～case文の例題として、指定した月の日数を求めるmdays()関数を作成します。mdays()関数では、図4-50のようにswitch～case文を使って、大の月「1,3,5,7,8,10,12月」の処理と小の月「4,6,9,11月」と「2月」の処理を分けています。2月の処理で使っているisleapyear()関数は、指定した年が閏年なら1を返し、閏年でなければ0を返す関数です。この関数は次項で作成します。

mdays（年，月）	指定した月の日数を返す

```
int mdays(int year,int month)
{
    int days;

    switch(month){                        ← 月によって処理を分岐させる
        case 1: case 3: case 5: case 7:   }
        case 8: case 10: case 12:         }← 大の月の日数は31日
            days=31;                      }
            break;
        case 4: case 6: case 9: case 11:  }← 小の月の日数は30日
            days=30;                      }
            break;
        case 2:                           }← 2月は閏年でなければ28日
            days=28+isleapyear(year);     }  閏年ならば29日
            break;
        default:
            printf("mdays(%d,%d):parameter error\n",year,month);
            days=31;
            break;
   }                 1から12まで以外の数値が与えられた時の処理
   return days;
}
```

図 4-50　mdays()関数

4.3.3　条件式

繰り返しや条件分岐の構文を使ったプログラムを実行するときに、実行の流れを決めるのは**条件式**です。本節では、条件式について徹底的に解説します。

比較式

条件式には文法上はどんな式でも書けるのですが、一般には**比較式**を書きます。主な比較式には表 4-1 のようなものがあります。

数学では「以上」や「以下」を「≦」や「≧」と書きますが、コンピュータのキーボードにはこの記号はないので、「<=」や「>=」と書きます。同様に「≠」を「!=」と書きます。

注目して欲しいのは、等しいかどうかを調べるために「＝」ではなく、「＝＝」を使うことです。C言語では「＝」が等号ではなく代入を意味するからです。「＝＝」のつもりで「＝」を使っても、左辺が変数であれば文法エラーとはならずに、代入演算が実行されてしまいます*5。最初のうちは間違えやすいので、注意してください。

*5 条件式の部分に代入演算を書くことによって、プログラムを短く書けることがあり（178 ページを参照）、C言語の特徴のひとつとなっています。しかし、初心者にとっては間違えやすい点なので、注意が必要です。処理系によっては、警告メッセージを出すものもあります。

優先順位	演算子	意味	使用例
高い ↑	<	より小さい	a<b　(aがbより小さい場合は真(1)、そうでない場合は偽(0))
	>	より大きい	a>b　(aがbより大きい場合は真(1)、そうでない場合は偽(0))
	<=	小さいか等しい	a<=b (aがbより小さいか等しい場合は真(1)、そうでない場合は偽(0))
	>=	大きいか等しい	a>=b (aがbより大きいか等いい場合は真(1)、そうでない場合は偽(0))
↓ 低い	==	等しい	a==b(aがbと等しい場合は真(1)、そうでない場合は偽(0))
	!=	等しくない	a!=b　(aがbと等しくない場合は真(1)、そうでない場合は偽(0))

表 4-1　主な比較式の種類

これまでの例題プログラムでも、図 4-51 のように比較式を条件式として使ってきました。

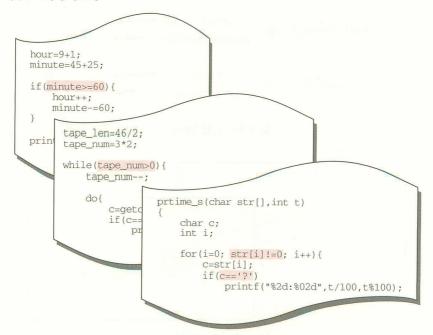



図 4-51　条件式を使った例題プログラム

式の値と条件

Ｃ言語では、すべての式が値を持ちます。比較式も例外ではありません。比較式は、図4-52のように比較した結果によって0か1の値を持ちます。

実をいうと、条件式の真偽は「0以外の値は真」、「0は偽」として扱われます。比較式は図のような値を持つので、真偽の判定に利用できるのです。

このことを応用すると、比較式に限らずどんな式でも条件式として使うことができます。たとえば、図4-53は例題プログラムの一部ですが、0かどうかを判定する条件式を短く書き直すことができます。

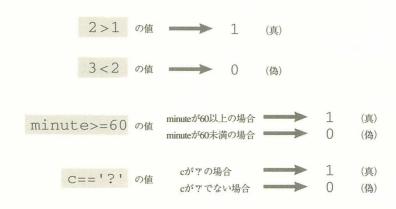



図4-52　比較式の値

```
prtime_s(char str[],int t)
{
    char c;
    int i;

    for(i=0; str[i]!=0; i++){
        c=str[i];
        if(c=='?')
            printf("%2d:%02d"
        else
            putchar(c);
    }
}
```

```
prtime_s(char str[],int t)
{
    char c;
    int i;

    for(i=0; str[i]; i++){
        c=str[i];
        if(c=='?')
            printf("%2d:%02d",t/100,t%100);
        else
            putchar(c);
    }
}
```

図4-53　式の値と条件

＋＋演算子や－－演算子を使った式も、やはり値を持ちます。C言語では、このことを利用したテクニックがよく使われます。次の図4-54は例題プログラムの一部ですが、－－演算子を使った式が値を持つことを利用して、図のように書き直すことができます。これは、デクリメントする前の変数tape_numの値が、そのまま「tape_num--」という式の値になることを利用したテクニックです。

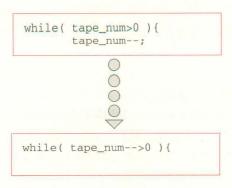



図4-54　－－演算子を使ったテクニック

なお、－－演算子や＋＋演算子の場合、演算子を変数の前につけるか後ろにつけるかで式の値が異なるので注意してください。図4-55のように、変数の

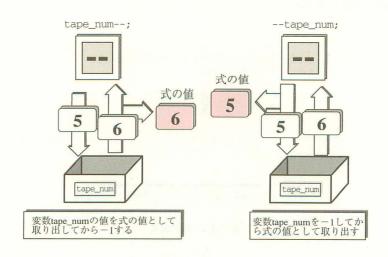



図4-55　－－演算子を使った式の値

後ろに演算子を付けるとデクリメントする前の値、前に付けるとデクリメントした後の値となります。

−−演算子を使った式が値を持つことを利用したテクニックなどは、C言語の特徴であり非常によく使われています。しかし、初心者のうちはわかりにくいので無理して使う必要はありません。わかりやすい方法でプログラムを書くことの方が重要です。

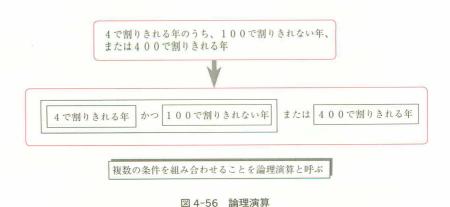
こうしたテクニックは、プログラム例などをいくつも読みこなしていくうちに次第に身についてくるでしょう。

論理演算子

条件は1つの式だけでは表せない場合も少なくありません。複数の条件を組み合わせて条件を判定するための**論理演算子**を解説しましょう。

本節での例題プログラムは、指定した年が閏年かどうかを判定するisleapyear()関数です。閏年は2月が29日まである年のことで、4年に一度オリンピックとともにやってきますが、正確には次の図4-56のような条件で決められています。

図のように、閏年は1つの条件では判定できず、3つの条件を組み合わせて判定しなければなりません。2つの条件を組み合わせて真偽を判定することを**論理演算**と呼びます。

図4-56　論理演算

論理演算には、図 4-57 に示す 2 種類の「論理演算子」を使います。「&&演算子」は、図-A のように 2 つの条件が「ともに成立すれば」真となります。「| | 演算子」は、図-B のように 2 つの条件のうち「どちらかが成立すれば」真となります。

条件 1 & & 条件2

A

		条件1	
		真	偽
条件2	真	真	偽
	偽	偽	偽

条件1と条件2がともに真の時だけ真

条件 1 | | 条件2

B

		条件1	
		真	偽
条件2	真	真	真
	偽	真	偽

条件1と条件2のどちらかが真ならば真

図 4-57 &&演算子と| |演算子

&（アンパーサンド）記号は「アンド」と読み、『～であり、かつ～』という意味です。|（バーティカルバー）は「オア」と読み、『～であるか、または～』という意味です。

論理演算子は必ず&&や| |のように、記号を 2 つ並べることに注意してください。記号 1 つの「&」や「|」は、それぞれまったく別な意味の演算子として存在します[*6]。&&のつもりで&と書いても、文法エラーとはならずに、&演算子として演算が行われてしまいます。

論理演算子を利用して作成した、閏年を判定する isleapyear()関数を次の図 4-58 に示します。

論理演算子には、優先順位があることに注意してください。「＊」と「＋」の関係のように、「&&」と「| |」では「&&」が優先されます。

＊6 & 演算子及び|演算子は、ビット単位の演算を表します。本書で解説すべき範囲を超えているので、他の書籍で学習してください。

isleapyear（年）	指定した年が閏年なら1を返し、閏年でなければ0を返す

```
int isleapyear(int y)
{
    if(y%4==0 && y%100!=0 || y%400==0)
        return 1;
    else
        return 0;
}
```

4で割った余りが0ならば4で割りきれることを利用する

図4-58　isleapyear()関数

プログラム部品ができあがったところで、日付を計算するメインプログラムを作成しましょう。図4-59に例題プログラムとその実行例を示します。日付の部分を変更して、いろいろ試してみてください。

```
main()
{
    int days1,days2,days3;

    days1=ydays(2000,9,3);
    days2=ydays(2000,1,18);
    printf("2000年9月3日は2000年1月18日の%d日後です。\n",days1-days2);

    days1=ydays(2001,9,3);
    days2=ydays(2001,1,18);
    printf("2001年9月3日は2001年1月18日の%d日後です。\n",days1-days2);

    days3=ydays(2001,3,28)+100;
    printf("2001年3月28日の100日後は");
    prdate(2001,days3);
    printf("です。\n");
}
```

1月1日から総日数の形にすれば日数の差を計算できる

1月1日から総日数の形にすれば日数の加算ができる

```
2000年9月3日は2000年1月18日の229日後です。
2001年9月3日は2001年1月18日の228日後です。
2001年3月28日の100日後は2001年7月6日です。
```

図4-59　日付計算プログラム

なお、このプログラムでは同じ年の日付しか計算できません。年を越えた日付の計算ができるようにするのは今後の課題としましょう。

〈本章で取り上げたプログラム〉　プログラム4-1

```
#include <stdio.h>

main()
{
    int t,tape_num;

    t = 1300; ······························スタート時刻を 13:00 にする
    tape_num = 2;

    tape(t,46,tape_num); ···················46 分テープを使った場合
    tape(t,60,tape_num); ···················60 分テープを使った場合
}

tape(int start_time, int tape_len, int tape_num)
{
    int t;

    printf("[ %d 分テープの場合 ]\n", tape_len);

    t = start_time;
    tape_len /= 2; ·························テープの長さは片面ごとなので tape_len÷2
    tape_num *= 2; ·························テープの数×2 が交換する回数

    while (tape_num >0) { ··················テープの数が正の間繰り返す
        tape_num--; ························テープの数を 1 つ減らす
        prtime(t);
        t = addclock(t,tape_len);
    }
}

prtime(int t)
{
    char c;

    do {
        c = getchar(); ·····················キーボードから 1 文字入力する
        if (c=='?') ························文字が「?」だったら時刻を表示
            printf("%2d:%02d",t/100,t%100);
        else ·······························「?」でなければ
            putchar(c); ····················文字をそのまま表示
    } while (c!='\n'); ·····················改行キーが押されるまで繰り返す
}

int addclock(int time1, int time2) ·········100 倍法で表した 2 つの時間を引数として受け取る
{
    int hour, minute;

    hour = time1/100 + time2/100; ··········時の加算
    minute = time1%100 + time2%100; ········分の加算

    if (minute >= 60) { ····················分が 60 以上であれば、時に繰り上げる
```

```
        hour++;
        minute -= 60;
    }

        return hour*100+minute; ………… 結果を100倍法で1つの数値にし、戻り値として返す

    }
```

〈本章で取り上げたプログラム〉 プログラム4-2

```
#include <stdio.h>

int tblA[] = { 930,1052,1205,1319,1427,1610 };

main()
{
    int fromAtoB; ………… A駅からB駅まで電車でかかる時間を入れる変数
    int start; ………… 出発時刻を入れる変数
    int i,t;

    fromAtoB = 236; ………… A駅からB駅まで2時間36分かかる
    start = 1000; ………… 10:00に出発する

    for (i=0;i<6;i++) ………… iを0から5まで繰り返す
        if (start < tblA[i]) ………… 出発時刻よりあとの電車があったら、繰り返しを中断
            break;
    if (i >= 6 ) { ………… iが6に達していたらもう電車はない
        printf("A 駅発の電車はもう終わりました.\n");
    } else {
        t = tblA[i]; ………… 次の電車の発車時刻
        printf("A 駅を %2d:%02d に出て ",  t/100,t%100);
        t = addclock(t,fromAtoB); ………… B駅に着く時刻を計算する
        printf("B 駅に %2d:%02d に着きます.\n",t/100,t%100);
    }
}

int addclock(int time1, int time2) ………… 100倍法で表した2つの時間を引数として受け取る
{
    int hour, minute;

    hour = time1/100 + time2/100; ………… 時の加算
    minute = time1%100 + time2%100; ………… 分の加算

    if (minute >= 60) { ………… 分が60以上であれば、時に繰り上げる
        hour++;
        minute -= 60;
    }

    return hour*100+minute; ………… 結果を100倍法で1つの数値にし、戻り値として返す
}
```

〈本章で取り上げたプログラム〉 プログラム4-3

```
#include <stdio.h>

main()
{
    int t;
    int tape_num;

    t = 1330;……………………………………スタート時刻を 13:30 にする
    tape_num = 3;

    tape_s(t,46,tape_num);
}

tape_s(int start_time, int tape_len, int tape_num)
{
    int t;

    t = start_time;
    tape_len /= 2; ……………………………テープの長さは片面ごとなので tape_len÷2
    tape_num *= 2; ……………………………テープの数×2 が交換する回数

    while (tape_num >0) {………………………テープの数が正の間くり返す
        tape_num--;……………………………テープの数を 1 つ減らす

        prtime_s("At ? , please exchange cassette tape\n",t);
        t = addclock(t,tape_len);
    }
}

prtime_s(char str[],int t)
{
    char c;
    int i;

    for (i=0;str[i]!=0;i++) }
        c = str[i];          } ……………引数の文字列を終わりまで 1 文字ずつ読む
        if (c=='?')
            printf("%2d:%02d",t/100,t%100); ……文字が「?」だったら時刻を表示
        else
            putchar(c); ……………………………「?」でなければ文字をそのまま表示
    }
}

int addclock(int time1, int time2) ……………100 倍法で表した 2 つの時間を引数として受け
{                                              取る
    int hour, minute;

    hour = time1/100 + time2/100; ……………時の加算
    minute = time1%100 + time2%100; …………分の加算

    if (minute >= 60) { ……………………………分が 60 以上であれば、時に繰り上げる
        hour++;
        minute -= 60;
    }

    return hour*100+minute; ……………………結果を 100 倍法で 1 つの数値にし、
                                               戻り値として返す
}
```

〈本章で取り上げたプログラム〉　プログラム4-4

```
#include <stdio.h>

int isleapyear(int y) ……………… 指定した年が閏年なら1を返し、閏年でなければ0を返す
{
    if ( y%4==0 && y%100!=0 || y%400==0 ) ……… 4で割った余りが0ならば4で割り切れることを利用する
        return 1;
    else
        return 0;
}

int mdays(int year, int month) ……………… 指定した月の日数を返す
{
    int days;

    switch(month) { ……………… 月によって処理を分散させる
    case 1: case 3: case 5: case 7: ……… 大の月の日数は31日
    case 8: case 10: case 12:
        days=31;
        break;
    case 4: case 6: case 9: case 11: ……… 小の月の日数は30日
        days=30;
        break;
    case 2: ……………… 2月は閏年でなければ28日
        days=28+isleapyear(year); ……… 閏年ならば29日
        break;
    default: ……………… 1から12まで以外の数値が与えられた時の処理
        printf("mdays(%d,%d):parameter error\n",year,month);
        break;
    }
    return days;
}

int ydays(int year, int month, int date) ……… 1月1日からの総日数を返す
{
        int days,i;

        days=0;
        for (i=1 ; i<month ; i++)
                days+=mdays(year,i); ……… 1月から前月までの日数を合計する
        days += date; ……………… 今月の日数を加える
        return days;
}

int prdate(int year, int days) ……………… 1月1日からの総日数に対応する日付を表示する
{
    int d,month,date;
    int i;

    for (i=1;i<=12;i++) {
        d = mdays(year,i); ……………… i月の日付を求める
        if (days <= d) { ……………… daysがi月の日数より少なければ
            month = i;   }
            date = days; } ……………… i月days日が求める日数
            break;
        }
        days -= d; ……………… 総日数からi月の日数を引く
```

```
    }
    printf("%d年%d月%d日",year,month,date);
}

main()
{
    int days1,days2,days3;

    days1 = ydays(2000,9,3);
    days2 = ydays(2000,1,18);
    printf("2000年9月3日は2000年1月18日の%d日後です。\n",days1-days2);

    days1 = ydays(2001,9,3);
    days2 = ydays(2001,1,18);
    printf("2001年9月3日は2001年1月18日の%d日後です。\n",days1-days2);

    days3 = ydays(2001,3,28)+100;
    printf("2001年3月28日の100日後は");
    prdate(2001,days3);
    printf("です。\n");
}
```

1月1日から総日数の形にすれば日数の差を計算できる

1月1日から総日数の形にすれば日数の加算ができる

第5章

コンピュータの仕組み

これまでの解説によって、コンピュータに処理手順を指示する命令書「プログラム」の書き方がだいたいつかめたと思います。前章までの文法を理解していれば、それだけで多くの種類の処理をプログラムとして記述することができます。しかし、それだけではC言語のパワーを最大限に活かすことはできません。C言語の機能をフルに活かすには、コンピュータがプログラムを実行する仕組みを知っていなければならないからです。

多くのプログラミング言語は、コンピュータの仕組みを知らなくてもプログラムが書けるように設計されています。変数や関数は実はコンピュータがプログラムを実行する仕組みに対応しているのですが、そのことを意識する必要はありません。ところが、次章で解説するポインタをはじとするC言語特有の機能は、コンピュータがプログラムを実行する仕組みに直結しており、しっかり理解していないと使いこなせないのです。

コンピュータの仕組みに直結した機能を持っているからこそ、C言語はパワフルなプログラミング言語としての評価を得ています。そうした機能を活かさなければC言語のパワーを満足に発揮することはできません。

本章ではC言語の機能を使いこなすために必要な最低限のコンピュータの仕組みを解説します。

本章で解説する項目

<table>
<tr><th></th><th>データ型</th><th>部品化機能</th><th>制御構造</th></tr>
<tr><td>5.1</td><td>コンピュータの内部構造
メモリ
ビットと数値</td><td></td><td></td></tr>
<tr><td>5.2</td><td>変数とメモリ
メモリとアドレス
変数宣言とメモリ
変数の型とメモリ
変数の型と数値</td><td></td><td></td></tr>
<tr><td rowspan="2">5.3</td><td></td><td></td><td>マシン語とは
演算子とマシン語</td></tr>
<tr><td colspan="2">ローカル変数と
グローバル変数</td><td></td></tr>
</table>

5.1

コンピュータの内部構造

コンピュータの内部構造

コンピュータの仕組みを知るために、まずはコンピュータの内部構造を簡単に解説しましょう。コンピュータの内部には図5-1のような回路基板があり、その上に多くのLSIと呼ばれるICチップが並んでいます。LSIはごく小さな半導体の上に、複雑な電気回路を集積したものです。コンピュータにはたくさんのICが使われていますが、これらは、その働きから図5-1のように大きく3つのグループに分けることができます。

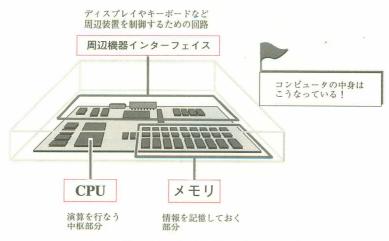



図5-1　コンピュータの内部構造

CPU（Central Processing Unit）はコンピュータの中枢となるLSIで、コンピュータ全体を制御し、情報処理そのものを行います。メモリはデータを記憶する部分です。それ以外の部分は画面やキーボードなど、外部の機器を接続するためのインターフェイス回路です。

この中でも最も重要なパーツはCPUです。ある意味では、CPUの能力がコンピュータの機能や性能を決めるといってもよいでしょう。CPU以外のパーツはいわば補助的な部品であり、コンピュータの中心的な機能である演算や情報処理はすべてCPUが行っているからです。このため、CPUのことを単独でコンピュータと呼ぶこともあります。

メモリ

コンピュータのパーツの中でもCPUの次に重要な役割を果たすのがメモリです。メモリは情報を蓄えておく記憶装置であることはよく知られています。キーボードやディスク装置などの周辺機器から読み込んだ情報は、メモリに蓄えて処理します。CPUが直接高速に処理できるのはメモリに記憶してある情報だけだからです。すべての情報はいったんメモリに記憶させてから処理します。

メモリの実体は図5-2のようなLSIチップですが、その仕組みを考える場合、図のように小さな箱がいっぱい詰まったものと思えばよいでしょう。LSIチップに電気的な信号を送ることによって、この箱に情報を記憶させたり、記憶させておいた情報を取り出したりすることができます。

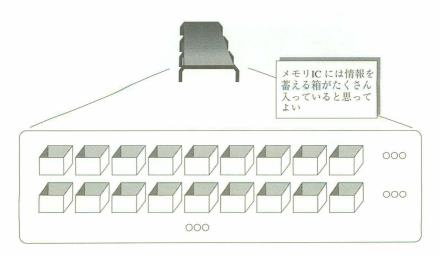



図5-2　メモリの仕組み

メモリの記憶単位であるこの箱のことを、**1バイトのメモリ**といいます。「バイト」という言葉は、メモリ容量やディスク容量を表す単位として耳にしたことがあるかと思います。1K(キロ)バイトといえば、約1000バイト、つまり約1000個の箱があることを意味しています。同様に1M（メガ）バイトならば図5-3のように約100万個の箱があることを意味しています。

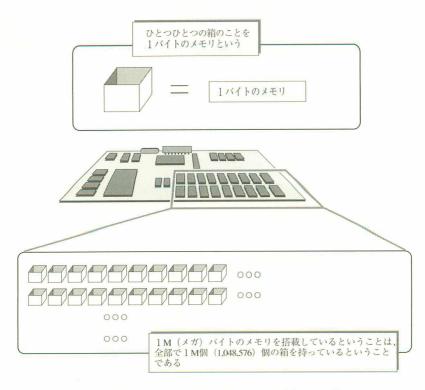



図5-3　1バイトのメモリ・1Mバイトのメモリ

コンピュータのK(キロ)

日常生活でもK(キロ)という言葉をよく使います。たとえば、1kgや1kmのように使っています。これはそれぞれ1,000g、1,000mのことであり、キロは1,000という数を表しています。ところが、コンピュータの世界では、1Kは1024という数を意味します。

これはなぜかというと、1024はちょうど2の10乗（2を10回掛けた数）だからです。コンピュータの原理は1と0という2つの状態の組合わせが基本です。われわれはふだん10進数を使っているので、10のべき乗(1, 10, 100, 1000, 1000…)をきりのいい数字と考えるのですが、2を基本的な単位とするコンピュータの世界では、2のべき乗（1, 2, 4, 8, 16, 32…)のほうが、きりのよい数字になるというわけです。

同様に、M(メガ)も1,000×1,000＝1,000,000ではなく、1024×1024＝1,048,576となります。

ビットとバイト

「ビット」という用語も、〝16ビット〟とか〝32ビット〟というかたちでやはりよく耳にすると思います。ビットもコンピュータの仕組みと密接に関わりのある言葉です。

図5-4を見てください。1バイトのメモリは、図のように8つの区画に分れています。この区画のひとつひとつを**ビット**と呼びます。

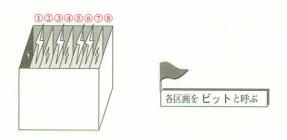



図5-4　ビット

メモリは、各ビットに電気を蓄えることによって情報を記憶します。各ビットは「電気を蓄えた状態」と「蓄えていない状態」という2種類の状態のうち、どちらかに設定できます。この機能によって「ある／ない」のどちらか、「YES／NO」のどちらかのように、二者択一の情報を記憶させることができます。「2種類のうちのどちらか」という情報は、最小の情報量といえるでしょ

う。このような意味で「ビット」はコンピュータで扱う『情報の最小単位』といえます。

ビットと数値

8個のビットに電気を蓄えることで、256種類の情報を記憶する仕組みを解説しましょう。1個のビットは2種類の状態のどちらかに設定できます。1バイトのメモリの中には8つのビットがありますから、とりうる状態は、2×2×2×2×2×2×2×2＝256種類になります。このようなビットの状態の組合わせを、**ビットパターン**と呼びます。図5-5のように、1バイトのメモリで256種類のビットパターンのうち1つを記憶できるというわけです。

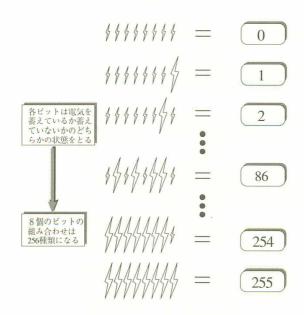



図5-5　ビットパターンと数値

メモリに数値を格納するということは、ビットパターンを格納することに相当します。しかしコンピュータでは、ビットパターンを図のように数値に対応させて、数値として処理します。実際にメモリに格納されるのはビットパターンですが、数値を格納していると考えるのです。

さらに、図5-6のように2バイトをまとめて1つの数値として扱うこともできます。2バイトは全部で16ビットですから、ビットパターンの種類はさらに256種類の2×2×2×2×2×2×2×2倍（＝256倍。8ビットの2倍ではないことに注意）で、65536種類になります。機種によってはさらに多くのバイト数をまとめて1つの数値として扱えるものもあります。

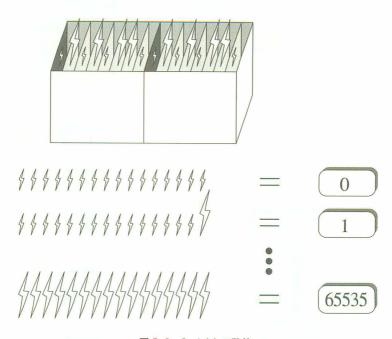



図5-6　2バイトの数値

C言語でプログラムを作成するときに、数値がビットパターンであることを意識する必要はほとんどありませんが、変数の型や大きさと深い関連があるので、ぜひ覚えておいてください。

また、コンピュータで扱う情報は、結局はすべてこのようなビットパターンであり、数値であることも理解しておいてください。住所や名前などの文字列も図形の形状や座標なども、すべて数値の組合わせに置き換えて処理します。

CPUとメモリ

前節で解説したように、メモリに数値を格納するというのは、メモリに8つの電気信号を送り込んで各ビットの状態を設定することに相当します。CPUとメモリは、次の図5-7のように**データバス**と呼ばれる信号線によって接続されています。CPUがメモリにデータを送る場合には、各ビットの電気的な状態をそれぞれ対応する信号線に出力します。メモリ装置はこの信号を読みとって、各ビットを対応する状態に設定します。

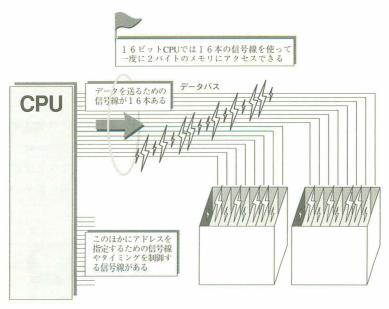



図5-7 CPUとメモリを結ぶ信号線

〝16ビットCPU〟とか〝32ビットCPU〟という言葉の16や32という数字は、この信号線の数を表しています*1。16ビットCPUの場合は信号線が16本あるので、一度に16ビット、つまり隣合った2バイト分のメモリからデータを読み出すことができます。さらに32ビットCPUでは一度に4バイト分のデータを読み出すことができるというわけです。

---

*1 厳密な定義ではなく例外もありますが、概念的にはこのように思っていてかまいません。

5.2

変数とメモリ

C言語のプログラムは、コンピュータの仕組みと実によく対応しています。コンピュータの仕組みのエッセンスをわかりやすいかたちに置き換え、プログラミングを容易にしたのがC言語であるといえるでしょう。本節では、前節で解説したコンピュータの仕組みが、C言語でどのように表現されているかを解説します。

メモリとアドレス

CPUとメモリの間で情報をやりとりすることを、**メモリをアクセスする**といいます。メモリをアクセスするためには、たくさんある箱の中から、情報を記憶させたい箱や情報を取り出したい箱を指定しなければなりません。そのために、たくさんあるメモリには、すべて図5-8のように通し番号が付い

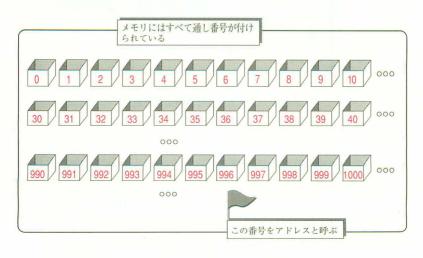



図5-8　メモリとアドレス

ています。CPUがメモリをアクセスするときは、メモリを特定するために、かならずこの番号を指定します。これは、あたかもメモリの住所を表しているようなものなので、この番号のことを**アドレス**と呼びます。

変数宣言とメモリ

C言語では、情報の入れ物を変数として用意し、情報を変数に入れて処理します。すでにおわかりのように、C言語の変数はそのままメモリに対応するもので、情報をメモリに格納して処理することを変数というかたちで表しているのです。

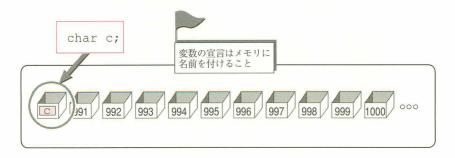



図 5-9　変数宣言

変数を宣言すると、図5-9のように情報の入れ物としてメモリが割り振られ、そのアドレスと変数の名前が対応付けられます。言い換えると、変数宣言はメモリに名前を付ける作業ということになります。

変数とメモリの対応は、C言語の処理系が適当なアドレスを自動的に割り当ててくれます。変数に情報を代入したり、変数の値を使って演算したりするプログラムは、その変数のアドレスのメモリに情報を書き込んだり、読み出したりする処理に変換されます。

変数の型とメモリ

char型の変数は、そのまま1バイトのメモリと対応します。char型の変数に値を代入するということは、1バイトのメモリにデータを書き込むことになります。char型は、主に文字を格納するために使うデータ型です。コン

ピュータ内部では文字を数値に対応させて処理します。英文字の場合は、256種類の数値で十分表せるので、char 型を1バイトのメモリに対応させているのです。

int 型の変数は、2バイトのメモリに対応します。int 型は主に整数を格納するために使いますから、256 種類では足りません。そこで 65536 種類の数値を表すことができるように、2バイトのメモリに対応させるのです。

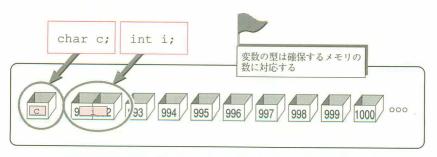



図 5-10　変数の型とメモリ

このように、変数の型によって1つの変数に対応するメモリの数が違います。変数の型を宣言するということは、ひとつには確保すべきメモリの数を指示していることになるのです。

変数の型と数値

変数の型には、必要なメモリの大きさを指示するだけでなく、もうひとつ大きな役割があります。それはビットパターンと数値との対応を決めることです。詳しくは次の6章で解説しますが、C言語の変数型のひとつに「unsigned int 型」（アンサインド・イント）があります。int 型も unsigned int 型も2バイトのメモリに対応しますが、図 5-11 のようにビットパターンと数値との対応が異なります。

int 型の変数では、負の数値を扱うことができますが、unsigned int 型の変数では負の数値は扱えません。しかし、int 型よりも大きな数値を扱うことができます。このように、変数型は、65536 種類のビットパターンに対応する数値の範囲を決めるためにあるのです。

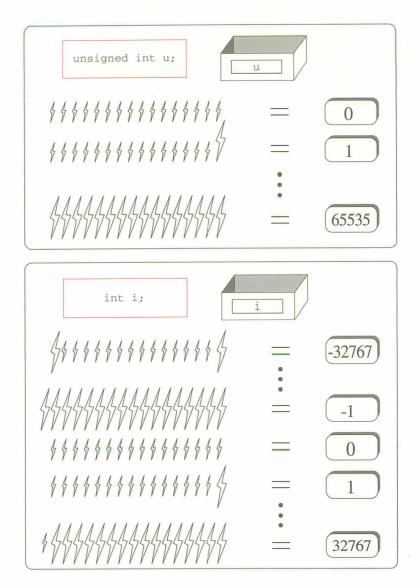



図 5-11 変数の型と演算

5.3 マシン語

マシン語とは

コンピュータの中で、実際に演算などの処理を行っているのはCPUです。CPUは、プログラムとして書かれた命令を次々に読み込みながら処理を実行していきます。

CPUが直接実行できる命令は、図5-12に示すようにC言語の演算子1つに対応するような単純な処理にすぎません。C言語のプログラムは、図のようにCPUが実行できる命令の列に変換することによって、はじめて実行できるようになります。

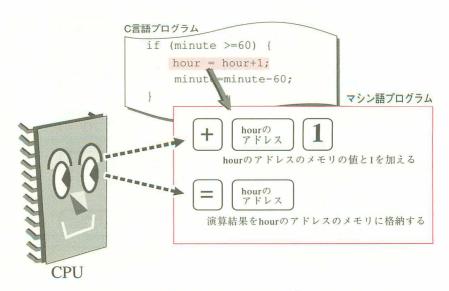



図5-12　C言語とマシン語

CPUが直接実行できる命令語のことを**マシン語**と呼びます。『CPU（マシン＝コンピュータ）が読むことのできる言葉』という意味です。

マシン語命令によってCPUが実行する処理は、図に示したように非常に単純なものです。C言語のプログラムも、このようなマシン語命令に変換しなければなりません。この変換作業が**コンパイル**です。

演算子とマシン語命令

C言語の演算子の多くは、マシン語の命令にそのまま対応しています。このため、「＋＋」や「＋＝」などの演算子を積極的に使うことによって、効率のよいプログラムを書くことができます。

たとえば、前の図5-12のプログラムは＋＋演算子を使うと、次の図5-13のように書くことができます。そうすれば、プログラムを短く書けるだけでなく、コンパイル後のマシン語命令の占めるメモリが節約され、実行にかかる時間も短縮されます。

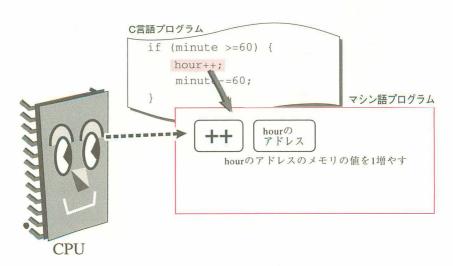



図5-13　演算子とマシン語命令

このようにC言語は、マシン語の命令に対応した多くの演算子を利用して効率のよいプログラムを作成できることが特長です。

C言語プログラムとマシン語プログラム

マシン語命令は、次の図5-14のようにビットパターンのかたちでメモリに格納しておき、CPUは、ビットパターンをメモリから順次読み込んで、その指示に従って処理を実行します。

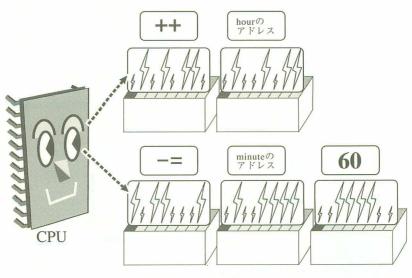



図5-14　マシン語命令

マシン語プログラムも、変数と同じようにメモリに割り当てられることがわかりました。したがって、C言語のプログラムは、コンパイル作業によって、次の図5-15のような実行可能なプログラムに変換されることになります。

変数に対応するデータ領域も、処理実行部に対応するマシン語プログラム領域も、ともにメモリに置かれることをよく覚えておいてください。

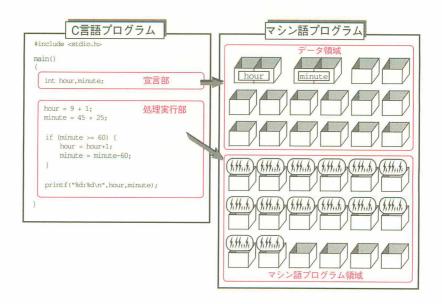



図 5-15 C言語プログラムとマシン語プログラム

もうひとつ注目してほしいのは、C言語プログラムのデータ宣言部と処理実行部以外の「部品化のための指示」の部分が、マシン語プログラムの中には存在しないことです。

部品化のための指示は、プログラム部品どうしを結び付けるためにコンパイラが処理する命令です。プログラム部品を結び付けてできあがった実行可能プログラムには、もはや必要ないのです。部品化のための指示は、いわば建築現場の足場のようなものだと思えばよいでしょう。プログラムを作成している最中には必要ですが、完成したプログラムには無用となります。

ローカル変数とグローバル変数

変数とメモリの関係について、もう少し知っておいてほしいことがあります。それはローカル変数とグローバル変数の違いです。

4.1 節で解説したように、関数の外側で宣言した変数はグローバル変数となり、どの関数からでも参照することができます。これに対して、関数の中で宣言したローカル変数は、その関数内でしか参照できません。グローバル変

数とローカル変数の大きな違いは、このような変数の見える範囲にあるのですが、もうひとつ重要な違いがあります。

図5-16は、4.2節の時刻表プログラムを実行しているときのメモリの様子を表しています。グローバル変数は、コンパイル時、つまりC言語プログラムから実行可能なプログラムに変換する時点でメモリ領域が確保され、変数に対応するメモリアドレスが決まります。

これに対してローカル変数は、図のように関数が呼び出されて実行を開始する時点ではじめてメモリ領域に割り当てられます。関数の引数も同様です。そして、関数の実行を終了すると、ローカル変数に使用されたメモリは最初の状態に戻されて、他の関数のローカル変数に再び割り当てられて使用されます。

つまり、ローカル変数は、関数の実行中だけ存在する変数なのです。したがって、変数の初期値は決まっておらず、また2回目の呼び出しであっても前回代入した値は残っていません。

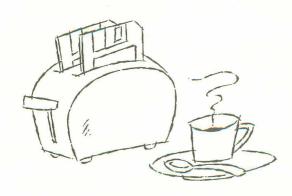
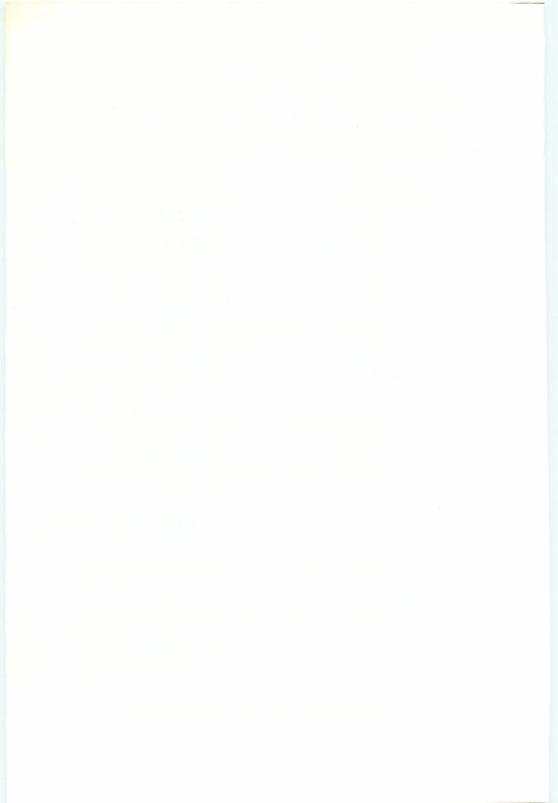



図 5-16　ローカル変数とグローバル変数

第6章

データ型のすべて

C言語を使いこなすポイントとなるのが「データ型」の使い方です。データ型の考え方は簡単なことのように思えますが、実はかなり奥が深いものです。なぜなら、処理する情報の種類によってデータ型をうまく選んだり組み合わせたりするのは、かなりの熟練を要するからです。

C言語のデータ型には他の言語にない特徴があり、コンピュータの仕組みを理解していないと使いこなせない面があります。前章でコンピュータの仕組みを解説したのは、データ型の考え方を理解してもらうために必要なことだからです。とくに、ポインタを理解するためには、コンピュータの仕組みについての理解が欠かせません。

本章では、これまでの解説の再確認を含めて、データ型を総合的に解説します。C言語のデータ型は非常にシンプルで、かつ応用しやすく設計されていますから、カギとなるところをしっかり押さえておけば自在に使いこなすのはそう難しくないでしょう。なかでも、初心者がつまずきやすいと言われるポインタについては、前章の解説を思い出しながら重点的に理解するようにしてください。

本章で解説する項目

<table>
<tr><th></th><th>データ型</th><th>部品化機能</th><th>制御構造</th></tr>
<tr><td>6.1</td><td>データ型の種類
基本データ型の種類
unsigned型
型変換(キャスト)
型の自動変換
char型とint型の関係
文字の処理
文字から数値への変換</td><td></td><td></td></tr>
<tr><td>6.2</td><td>ポインタとは
ポインタ型変数
&演算子
＊演算子
ポインタと配列
ポインタのインクリメント</td><td></td><td></td></tr>
<tr><td rowspan="2">6.3</td><td>複合データ型(構造体)
構造体の宣言
タグ
メンバ</td><td></td><td></td></tr>
<tr><td colspan="2">構造体と関数
構造体へのポインタ
－＞演算子</td><td></td></tr>
</table>

6.1

データ型の種類

最初に、4章で解説したデータ型の種類について復習しておきましょう。
データ型には、次の図6-1のように、大きく分けて3つの種類があります。

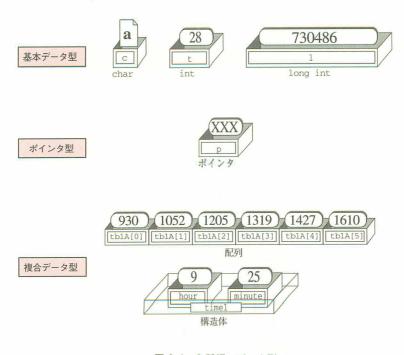
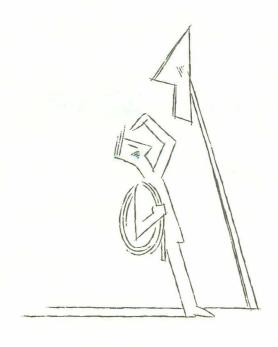



図6-1　3種類のデータ型

基本データ型は、数値を格納するためのデータ型です。コンピュータの情報処理の基本は、情報を数値に置き換えて処理することにあります。基本データ型にはいくつかの種類があり、情報の種類に応じて最適なものを選んで利用します。

ポインタ型は、メモリのアドレスを格納するためのデータ型です。5章で、変数名はメモリのアドレスに付けた名前であることを解説しました。アドレスの代わりに変数名を使うことで処理をわかりやすく書くことができるのですが、逆に変数名がアドレスであることを利用すると、スマートに処理できる場合があります。

複合データ型は、データ型を組み合わせて新しいデータ型を作る機能です。複合データ型には配列や構造体、共用体があります。

6.2

基本データ型

6.2.1 基本データ型のすべて

コンピュータはあらゆる情報を処理することができますが、その秘密はすべての情報を数値に置き換えて処理することにあります。逆にいえば、コンピュータは、数値しか扱うことはできません。

数値に置き換えた情報は、基本データ型の変数に格納して処理します。本節では、C言語に用意されている数値型について、その使い分けや型変換の方法を解説します。

基本データ型の種類

コンピュータが扱える数値は、決して無限ではありません。8桁の電卓では9桁の数値を計算できないように、int型の変数に入れておける数値には限界があります。147ページで解説したように、コンピュータ内部では、ビットパターンを数値に対応させて処理します。したがって、ビットパターンの種類の数だけの数値を扱うことができます。int型の変数では、下の図6-2のように、−32767から32767までの数値を扱うことができます。

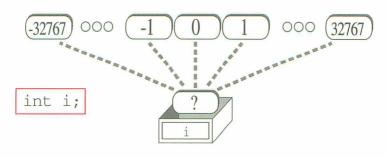



図6-2　int型の変数で扱える数値の範囲

図 6-3 のように、int 型の変数を使った演算の結果がこの範囲に納まらない場合は、**オーバーフローする**（『あふれ出す』という意味）といって、正しい演算結果が得られません。int 型の変数を使ってプログラムを作成するときには、この点に十分注意しておかなければなりません。

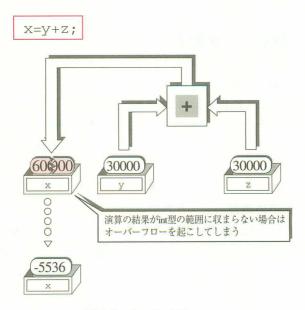



図 6-3　オーバーフロー

広い範囲の数値を扱うためには、それだけ多くのメモリを必要とします。CPU とメモリは、データバスで接続されていることを思い出してください。データバスの容量を越える情報をやりとりするためには、何回かに分けて転送しなければなりません。したがって、数値の範囲を広げると、処理に要する時間が増えることになります。

C 言語では、処理速度を低下させない範囲で、最も広い範囲の数値を扱える変数のメモリサイズを int 型としています（基本的にはデータバスのサイズと一致する）。処理スピードを犠牲にしてでも広い範囲の数値を扱うために、さらにいくつかのデータ型が用意されています。それが、図 6-4 に示す long int 型や double 型です。

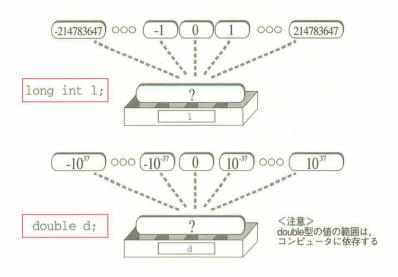



図 6-4　long int 型と double 型

逆に、int 型よりももっと狭い範囲で十分な場合のために、char 型や short int 型などのデータ型が用意されています*1。これらのデータ型を選択すると、変数として使用するメモリ容量を節約することができます。

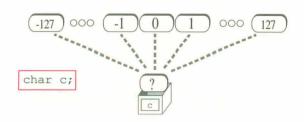



図 6-5　char 型

unsigned 型

165 ページで解説したように、同じメモリサイズでも、ビットパターンと数値の対応には、2 つの方法があります。ビットパターンを正の数だけに対応さ

＊1　MS-DOS 用の C 言語処理系の場合、int 型と short int 型は同じ型を表します。

せる方法と、正負両方の数に対応させる方法です。

int 型や long int 型では、正負両方の数に対応させる方法を使います。これに対し、正の数だけに対応させる方法を使うのが、unsigned　int 型や unsigned char 型です。unsigned（アンサインド）は『符号なし』という意味で、正の数だけを扱うためのデータ型です。unsigned が付くデータ型では、図 6-6 のように、負の数が扱えない代わりに、正の数で広い範囲の数値を扱うことができます。

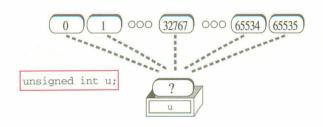



図 6-6　unsigned int 型

基本データ型の役割は、扱える数値の範囲を選択することです。通常の処理では、int 型を選んでおけば間違いありませんが、もっと広い範囲の数を扱いたいときは、多少効率が悪くなることを覚悟しても、int 型よりも広い範囲の数値を扱えるデータ型を選択します。また、配列など、たくさんの数値を扱う際にメモリ容量を節約したい場合には、char 型などを選択します。

ここで、まとめとして、すべての基本データ型について扱える数値の範囲を表 6-1 に示しておきます。なお、long int 型、short int 型、unsigned int 型は、int を省略して long 型、short 型、unsigned 型とすることができます。

型名	種類	範囲
char	整数	-127〜127
unsigned char	正の整数	0〜255
short int	整数	-32767〜32767
unsigned short int	正の整数	0〜65535
int	整数	-32767〜32767
unsigned int	正の整数	0〜65535
long int	整数	-2147483647〜2147483647
unsigned long int	正の整数	0〜4294967295
float	実数	約 1.18E-38〜約 3.40E+38
double	実数	約 2.23E-308〜約 1.80E+308
long double	実数	約 3.36E-4932〜約 1.19E+4932

・MS-DOS 用の主なC言語処理系における値
・処理系によっては、char 型の範囲が unsigned char 型と同じ場合もある

表 6-1　基本データ型で扱える数値の範囲

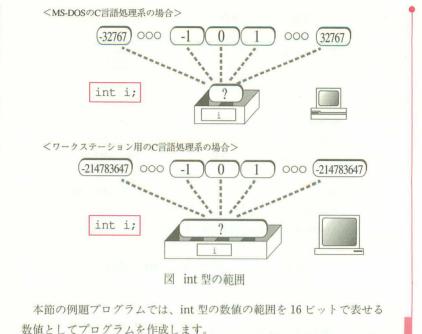
CPU の種類と int 型のサイズ

int 型で表せる数値の範囲は、C 言語の言語仕様として決まっていません。その理由は、最も効率よく処理できる変数のメモリサイズは、CPU の種類によって違うからです。C 言語では、CPU の種類ごとに最適なメモリサイズを int 型に割り当てることにしているので、どの CPU でももっとも効率のよいプログラムを作成することができます。

MS-DOS 用のほとんどのC言語処理系では、図のように、int 型の数値の範囲を−32767 から 32767 までとしています。MS-DOS の動作する 8086CPU は 16 ビット CPU であり、int 型の変数を 16 ビットのメモリに割り当てるのが最も効率がよいからです。たとえ 80386 などの 32 ビット CPU が使われている機種でも、MS-DOS を使っている限りは、プログラムの互換性のため int 型変数のサイズは 16 ビットとなります。

これに対し、ワークステーションなど 32 ビット CPU を使った機種では、int 型の変数を 32 ビットのメモリに割り当てています。UNIX 用のプログラムを MS-DOS に移植するときなどには、この違いに気をつける必要があります。

図　int 型の範囲

本節の例題プログラムでは、int 型の数値の範囲を 16 ビットで表せる数値としてプログラムを作成します。

型変換（キャスト）

本節で、先の 4.3 節で宿題となっていた日付計算プログラムを、年を越える場合でも計算できるように改良しましょう。前のプログラムでは、その年の 1 月 1 日からの総日数を求めてから計算を行いましたが、今回は西暦 1 年 1 月 1 日からの総日数を求めることにします。

暦の数え方は文明の進歩とともに変化していますから、現在の暦だけを使って求めることはできないのですが、便宜上、現在と同じ暦が西暦 1 年 1 月 1 日から使われていたとして計算します。

計算してみればわかりますが、この値は 165 ページの図 6-2 に示した int 型の数値の範囲を超えてしまいます。たとえば、21 世紀のはじまる 2001 年 1 月 1 日は、730485 日目となり、long int 型を使わなければ計算できません。

図 6-7 に、西暦 1 年 1 月 1 日からの総日数を求める ldays()関数のプログラムを示します。この ldays()関数は、long int 型のデータを返す関数になっています。

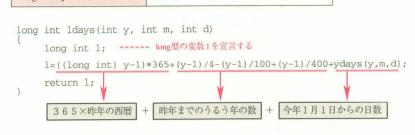



図 6-7 ldays()関数

このプログラムのように、計算に使用する数値がすべてint型なのに結果がlong int型になるという場合には、ちょっとした工夫が必要になります。

図6-8を見てください。式中の各変数は、int型の範囲に十分納まる値しかとらないのですが、図のように、演算の結果はlong int型の範囲に納まりません。結果をlong int型の変数に代入するようにしていますが、代入前の演算の過程でオーバーフローしてしまうので、正しい結果は得られないのです。

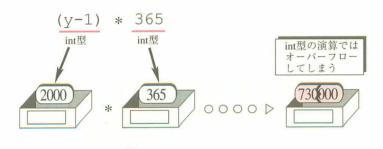



図 6-8 オーバーフロー

このような場合には、**キャスト**と呼ばれる演算を行います。キャストは型変換を行う演算で、型名を( )で囲んだ**キャスト演算子**を使います。ldays()関数では、図6-9のように、「(long int)」というキャスト演算子で、変数の型をlong int型に変換しています。こうしておけば、変数がlong int型であるとして演算が行われるので、オーバーフローは起こりません。

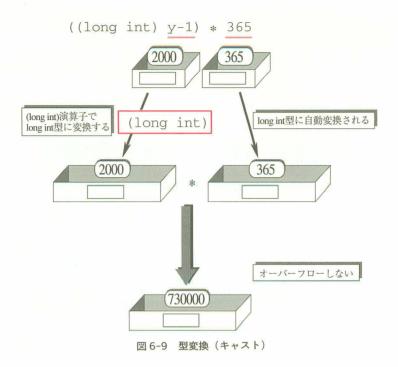
図 6-9　型変換（キャスト）

型の自動変換

図 6-10　自動型変換

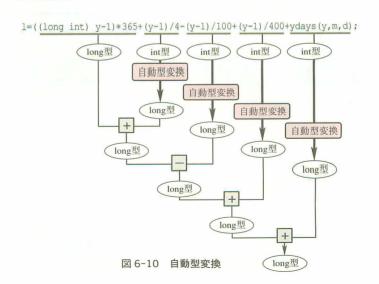



図6-10のように、異なるデータ型の変数どうしで演算を行う場合には、自動的にメモリサイズの大きい方のデータ型に型変換されてから演算されます。このような型変換は自動的に行われるので、ほとんど意識する必要はありません。

日付計算プログラム

最後に、プログラム6-1の残りの部分と、実行例を示します。年を越えた日付の差を正しく計算できていることを確かめてください。

```
main()
{
    long int days1,days2;

    days1=ldays(2000,3,28); ←── 西暦1年1月1日から2000年3月28日までの総日数を求める
    days2=ldays(1999,12,5); ←── 西暦1年1月1日から1999年12月5日までの総日数を求める
    printf("2000年3月28日は1999年12日5日の%ld日後です.\n",days1-days2);

    days1=ldays(2001,1,1);  ←── 西暦1年1月1日から2001年1月1日までの総日数を求める
    days2=ldays(1964,3,28); ←── 西暦1年1月1日から1964年3月28日までの総日数を求める
    printf("2001年1月1日は, 私が生まれてから%ld日目です.\n",days1-days2);
}
```

long int 型の数値を表示するには%ldを使う（Appendix 3参照）

図6-11　日付計算プログラム

```
2000年3月28日は1999年12月5日の114日後です.
2001年1月1日は, 私が生まれてから13428日目です.
```

図6-12　日付計算プログラムの実行例

6.2.2　情報と数値

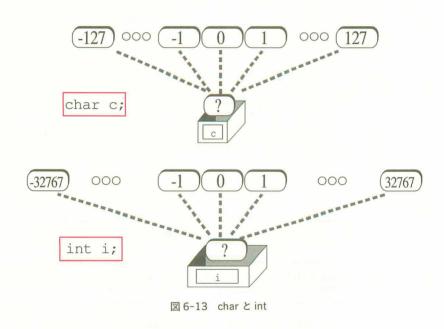
情報を数値に対応させて処理する典型的な例が、「文字」の処理です。コンピュータ内部で、文字をどのように処理しているのかを詳しく解説します。

char 型と int 型の関係

char 型は、文字を格納するための変数型として紹介しました。しかし、す

でに解説したように、char 型も int 型と同じ基本データ型の一種で、char 型の変数に数値を代入することもできますし、int 型変数と同じように演算を行うこともできます。

char 型と int 型の違いは、図 6-13 に示すように変数の大きさにあります。char 型の変数は 1 バイトと小さいので、−127 から 127 までの範囲の数値しか入れることができません。これに対して、 int 型の変数は 2 バイトなので、−32767 から 32767 までの数値を入れることができます。つまり、格納できる数値の範囲が違うのです。

図 6-13　char と int

しかし、文字を表す場合アルファベット文字は大文字、小文字、記号類を合せても数十種類しかないので、1 バイトで表せる範囲の数値（char 型）で十分なのです（日本語文字に関しては、180 ページのコラム参照）。

文字の処理

int 型は整数型で char 型は文字型というこれまでの解説は、主な使い方を意味したもので、コンピュータ内部では文字と数値は同じものです。図 6-14

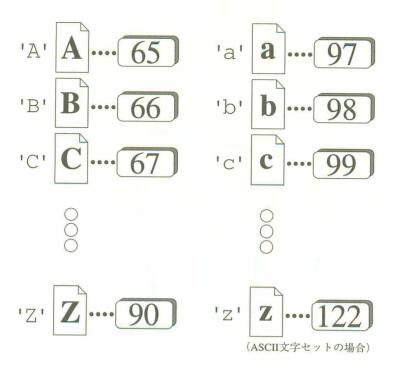



図 6-14　文字と数値

に示すように、文字にはすべて番号が振られています。コンピュータ内部では、この番号を使って文字を処理します。

文字を扱うためには、文字に対応する数値を扱うわけですが、だからといって対応する数値を覚える必要はありません。それは、C言語では、「'a'」と書くことで「文字a」に対応する数値を書いたことになるからです。ですから、これまでのプログラムでは文字を処理していたつもりでも、実は次ページの図6-15のように数値を処理していたのです。

「" "」で囲んだ文字列は、図のように数値の列として処理されます。そして、文字列の末尾には、数値の0を置くことを思い出してください。0は、どの文字にも対応しない数値ですから、文字列の末尾を示すマークとして使えるのです（Appendix 7の文字キャラクタセット一覧参照）。

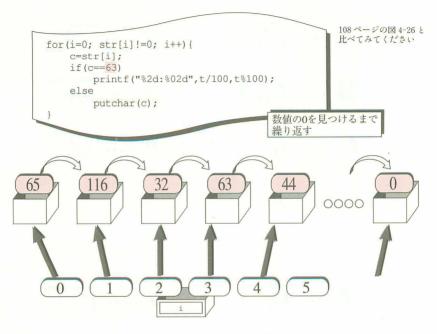



図 6-15　文字の処理

文字から数値への変換

キーボードのキーを押すと、キーに刻印されている文字が入力されます。すなわち、キーに刻印されている文字に対応する数値が入力されるのです。ですから、「1」というキーを押すと'1'という文字、すなわち、数値「49」が入力されます。

では、「1」のキーが押されたら数値1として処理するにはどうすればいいでしょう？

文字に対応する数値は、たとえば、「0」に対応する数値は48、「1」は49、「2」は50というように、大きな数字になるにつれて大きくなります。これを利用すると、図6-16のように'0'を引くことで、数字を数値に変換することができます。

また「9」、「4」と、続けてキーを押しても数値の94が入力されるのではなく、'9'と'4'という2つの文字が続けて入力されるのですから、キーボードからの入力で計算を行うには、「9」、「4」という2つの文字から94という数値

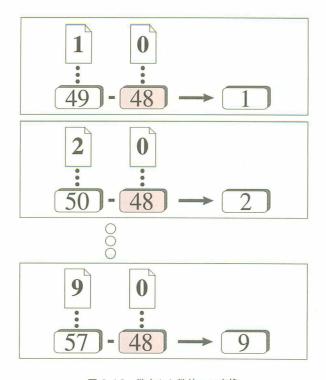



図 6-16　数字から数値への変換

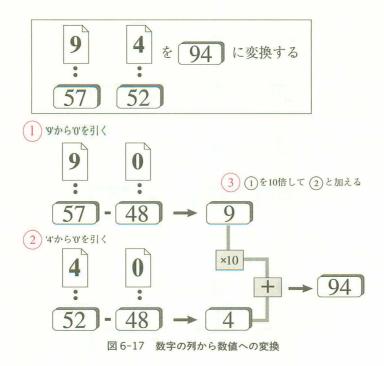
に変換しなければなりません。

その方法を表したのが、次ページの図 6-17 です。最初の数字を数値に変換し、それを 10 倍し、次の数字を数値に変換して加えます。数字の桁数が増えても、同様の処理を繰り返せば数値に変換することができます。

本節の例題プログラムは、この方法を利用した時間電卓プログラムです。このプログラムは、キーボードから入力した時間の加算と減算をします。

なお、数字の文字列から数値への変換処理は、ライブラリ関数の中に atoi() 関数として用意されています。こうした基本的な処理は、多くの場合ライブラリ関数の中から見つけることができるので、面倒なプログラムをわざわざ作る必要はありません。ライブラリ関数のマニュアルなどで、欲しいプログラムを探して、どんどん利用してください。

ここでは、文字と数値の関係をよく理解してもらうため、atoi()関数を使わ

図 6-17　数字の列から数値への変換

ずに以上のような処理を行うプログラムを作成しました。

代入式の値

図 6-18 に時間電卓プログラムを示しますが、このプログラムでは代入式が式の値を持つことを利用しています。「c = getchar()」という式は変数 c に getchar()の返す値を代入しますが、同時にその値が式全体の値にもなります。そこで、この式の値を'\n'と比較しているのです。

ここで注意してほしいことは、＝演算子よりも!＝演算子の方が優先順位が高いので、「c = getchar()!='\n'」と書くと「getchar()!='\n'」という式の値を変数 c に代入するという意味になることです。かならず図のように「(c = getchar())!='\n'」としなければなりません。

同様に「time = num = 0;」という式では、「num = 0」で変数 time に 0 を代入すると同時に「num = 0」という式の値 0 を変数 time に代入しています。次の図 6-19 にその実行例を示します。

```
calc(int time,int num,char ope)
{
    switch(ope) {
    case '+':
        time=addclock(time,num);
        break;
    case '-':
        time=subclock(time,num);
        break;
    default:
        time=num;
        break;
    }
    return time;
}

int main()
{
    char c;  ……入力された文字を格納する変数
    char ope;……1つ前の演算記号を記録しておく変数
    int num;……文字から数値への変換途中結果を格納する変数
    int time;……計算結果を格納しておく変数

    time=num=0;
    ope=' ';

    while ((c=getchar())!='\n') {
        if (isdigit(c))
            num=num*10+(c-'0');
        else if (c=='+') {
            time=calc(time,num,ope);
            ope=c;
            num=0;
        } else if (c=='-') {
            time=calc(time,num,ope);
            ope=c;
            num=0;
        } else if (c=='=') {
            time=calc(time,num,ope);
            printf("%2d:%02d\n",time/100,time%100);
            time=num=ope=0;
        } else
            break;
    }
}
```

addclock()関数、subclock()関数は、章末のリストを参照

（else if 〜 time=num=ope=0; の部分）……1つ前の式を計算し，次の数値が入力された時に計算するために演算記号をとっておく

図 6-18　時間電卓プログラム

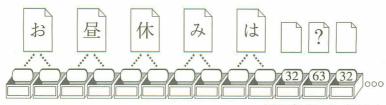
```
945+125=
11:10
```

図 6-19　時間電卓プログラムの実行例

日本語の処理

アルファベットや記号は全部で数十種類しかないので、char 型の変数、すなわち1バイトで表せる数値の範囲に十分納まります。しかし、数千から数万種類に及ぶ漢字やかなは、とても1バイトで表せる数値の範囲には納まりません。そこで、日本語文字には、2バイトの数値で表せる番号を割り当てます。文字と番号との対応は、JIS 規格として定められているのですが、一般には JIS 規格に多少の変更を加えた方式が使われています。たとえば、MS-DOS では、シフト JIS 方式と呼ばれる方式が使われています。

また、日本語文字（全角文字）は、アルファベット（半角文字）のように'あ'のような記法を使うことができません。ただし、「" "」で囲んだ文字列の中には日本語を使うことができます。プログラム中で日本語文字を含んだ文字列を使うと、図のように日本語文字1文字について2バイトの数値が割り当てられます。

"お昼休みは？からです。"

（7章のプログラムの一部です．）

本書の例題プログラムでは、日本語文字について対応していませんが、本来は日本語とアルファベットが混在している文字列を正しく処理するために、日本語文字とアルファベット文字を区別して処理しなければなりません。MS-DOS 用のC言語処理系では、こうした処理のために日本語処理用のライブラリ関数が用意されています。詳しくはマニュアルや他のC言語の書籍を参照してください。

6.3 ポインタ

C言語の機能の中でも、最も習得することが難しいとされるのが「ポインタ」です。しかし、ポインタは別に難しいものではありません。コンピュータの仕組みを、素直にしかもエレガントにC言語の機能として取り入れたにすぎないのです。

5章で解説したようなコンピュータの仕組みを理解していれば、簡単に使いこなすことができるでしょう。

6.3.1 ポインタとは

学校の授業を思い出してください。図6-20のように、先生が手に定規を握りしめていっしょうけんめい説明しています。黒板に書いた授業の要点を定規で指示しながら解説しています。

このように定規で「指す」ことを英語で「ポイント（point）」するといいます。ある地点を指し示すことです。

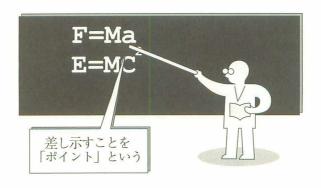



図6-20　ポイントする

C言語でポインタと呼ばれるのは、**ポインタ型の変数**のことです。ポインタ型の変数とは、図の先生のように「ポイント」する変数です。

ポインタの指し示すものは、次の図6-21のように他の変数です。他の変数を「ポイント」するので、「ポインタ(pointer)」というのです。

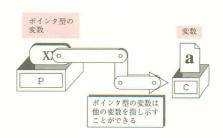



図6-21　ポインタ型の変数

ポインタとアドレス

前章でコンピュータの仕組みを解説したので、すでにお気付きだと思いますが、ポインタには他の変数のアドレスを入れます。図6-22のように、ポインタに変数のアドレスを入れることで、その変数を指し示すことができるのです。

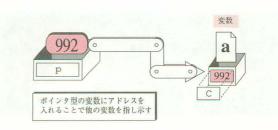
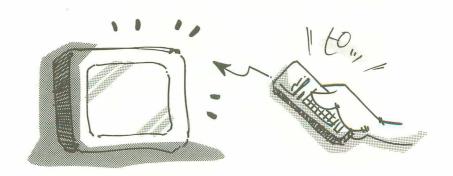



図6-22　ポインタ型変数とアドレス

ポインタの役割

ポインタは、指し示している変数の値を取り出したり、値を代入したりす

るために使います。ポインタの指している変数が、たとえその関数から見えない他の関数内の変数であっても、ポインタを使えば操作することができます。あたかもリモコンでテレビのチャンネルを操作するように、離れたところから自在に操作できるのです。

文法的な解説をする前に、ポインタを使って効率的な処理を実現できる場面を挙げて、具体的に解説しましょう。

(1) 配列を高速処理する

ポインタ型変数を使うと、処理を高速化できる場合があります。とくに配列の処理は、ポインタを使うことによって高速な処理が可能になります。次ページの図 6-23 に示すように、配列要素へのアクセスには、かならず配列先頭のアドレスに要素の番号を加えるという演算が必要になりますが、ポインタを使えばこの演算は不要になり、高速な処理が可能なのです。

(2) こみいった変数の受け渡しをする

4 章の 92 ページで解説したように、変数を引数に指定して関数を呼び出すときには、その値を関数の引数変数にコピーして渡します。したがって、引数変数をいくら変更しても、呼びだした側の変数を変更することはできません。

しかし、呼びだした側の変数を変更したい場合もあります。たとえば、関数から複数の値を返したいときなどがそうです。関数は戻り値を 1 つしか返せませんが、複数の値を返したいときもあるでしょう。

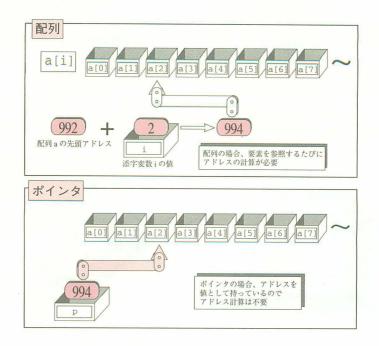



図6-23　配列とポインタ

また、配列や構造体など、複合データ型の変数を関数の引数として渡す場合を考えてみてください。複合データ型の変数は、たいへん大きなメモリサイズを持つことがあり、関数の引数変数に変数の値をコピーしていると、引数変数に割り当てるメモリの量や、コピー処理にかかる時間がバカになりません。

このように変数の値ではなく変数そのものを渡したい場合には、図6-24のようにポインタを使います。変数を指し示すポインタを渡すことで変数そのものを渡すことになるのです。

(3) 構造体と組み合わせる

186ページの図6-25は、7.2節で解説するスケジュール表を処理するプログラムのデータ構造を表したものです。スケジュール表に順序よく予定を書いておいても、新しい予定が入ってきたり、キャンセルされることもありま

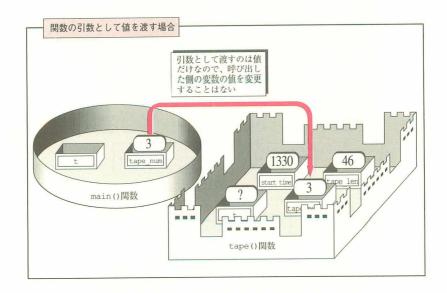
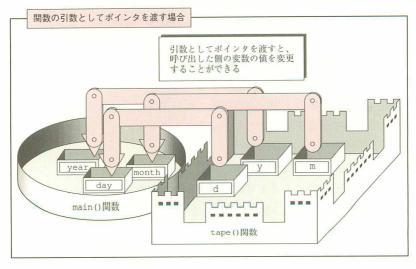



図 6-24　ポインタと引数の受け渡し

す。そのたびに予定を順序通りに書き直すのは面倒ですから、図 6-25 のような矢印で順序を書き表すことになるでしょう。

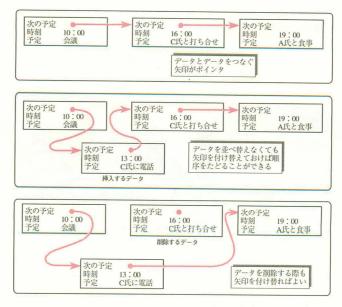



図 6-25　ポインタによる構造体の連結

このように追加や削除を矢印の付け換えとして表現できる情報は、構造体とポインタを組み合わせると、実にうまく処理することができます。新しい予定が入ったら、情報の中身を入れ換えるのではなく、ポインタを付け換えることによって順番を変更することができるのです。

(4) 処理中にメモリを割り当てる

変数として割り当てるメモリには、ローカル変数領域とグローバル変数領域の2種類があることは、5章で解説しました。実はこれ以外に、もう1つ、**ヒープ領域**と呼ばれる変数領域があります。

あらかじめ用意しておいた変数以外に、プログラム実行中に情報を格納する変数が必要になる場合、ヒープ領域のメモリを割り当てて利用します。ヒープ領域のメモリにはあらかじめ変数名を付けておくことができないので、図6-26のようにポインタを使ってアクセスします。具体的な方法は、7章で解説します。

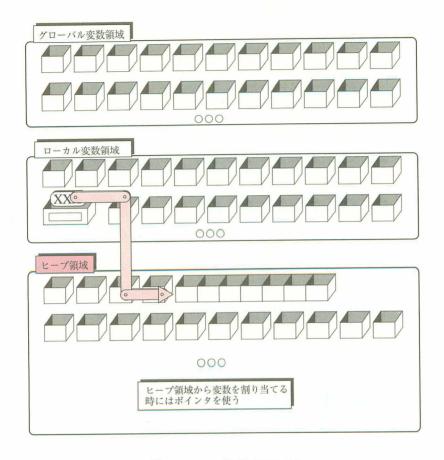



図 6-26　ヒープ領域のアクセス

(5) インターフェイス装置を制御する

5章で解説したコンピュータの内部構造を思い出してください。コンピュータの内部にはCPUとメモリ以外に、周辺装置を制御するためのインターフェイス回路があります。

インターフェース回路には、メモリと同じように装置ごとにアドレスが割り当てられており、周辺装置からデータを読み出すには、インターフェイス回路のアドレスを指定して、データを読み出します。逆に、周辺装置を動かしたり、データを書き込むには、インターフェイス回路のアドレスを指定し

て、信号を送ります。

ポインタにインターフェイス回路のアドレスを入れると、図6-27のように、インターフェイス回路をあたかも変数であるかのように処理することができます。変数の値を取り出したり、変数に代入することと同じような処理で、インターフェイス装置を直接制御することができるのです*2。

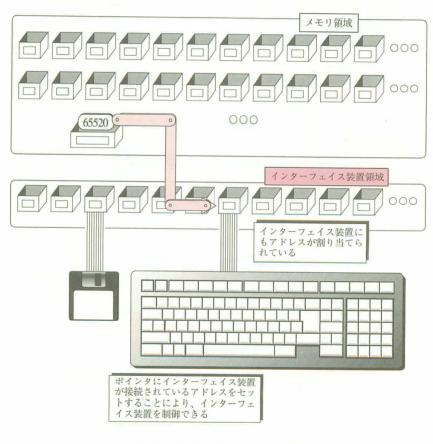



図6-27　インターフェイス装置の制御

＊2　8086系CPUを使ったコンピュータでは、ビデオメモリをポインタを使って制御することは可能ですが、他のインターフェイス装置を制御することは、一般にはできません。インターフェイス装置が、メモリアドレスではなく、I/O(アイオー)アドレスを使ってアクセスする仕組みになっているからです。

一般的なプログラムではこのような処理は必要ありませんが、オペレーティングシステムやデバイスドライバを作成する際には必要な機能です。また、パーソナルコンピュータでは、画面を制御するためのビデオメモリを直接制御して、高速な画面操作を実現することもあります。

ポインタの危険性

C言語以外のプログラミング言語にも、(2)、(3)、(4)の機能を持っているものがあります。しかし、各機能に専用の書式が用意されていて、他の用途に用いることはできません。(1)と(5)の機能は、他の言語にはあまり見られないC言語に特有の機能です。C言語のポインタは、これらの機能の本質的な部分を素直な形で表現したものです。コンピュータの仕組みを、そのまま利用した機能であるために応用の幅が広いのです。

その代わり、使い方を誤ると非常に危険な一面を持っています。指し示す先を間違えると、他の変数を変更してしまう可能性があるのです。

次ページの図6-28のように、ポインタの値が誤って設定されると、変数領域だけでなくマシン語プログラム領域やシステムの領域に書き込んでしまう可能性があります*3。

ポインタでのプログラムミスは重大な結果を生む可能性があるので、十分注意してプログラミングするようにしてください。

＊3　UNIXなどのオペレーティングシステムでは、不正なメモリアクセスをチェックするメモリ保護機能があり、実行中のプログラムを強制的に停止します。パーソナルコンピュータ用のオペレーティングシステムであるMS-DOSなどの場合は、メモリ保護機能がないので、システム全体が停止したり、暴走してしまったりします。

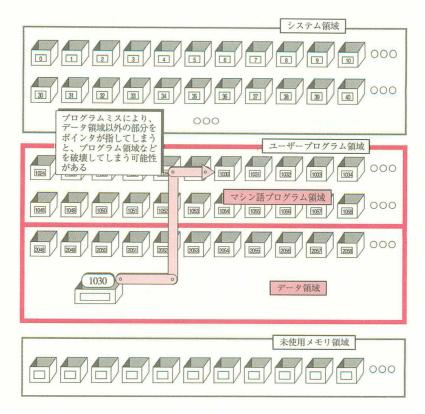



図6-28 ポインタの危険性

6.3.2 ポインタの使い方

ポインタ型変数

［書式］型名 *変数名 ［= 初期値］;

いよいよポインタの文法的な解説に入ります。ポインタ型の変数を宣言するには、上の書式に示したように、変数名の前に*（アスタリスク）を付けます。「*」は掛け算の演算子でもありますが、この場合は、ポインタであることを示す記号になります。型名は指し示す先の変数の型を表していて、たとえば「char *p;」という宣言は、図6-29のように『char型の変数を指し示

す、ポインタ型変数p』を宣言するという意味になります。ここで誤解しないように気を付けてほしいのは、型名は「char *」であって変数名は「p」であるということです。「char *型の変数pを宣言する」といってもよいのですが、いいにくいので「charを指すポインタ型」と呼べばよいでしょう。

他の変数と同様に、ポインタ型変数は宣言しただけでは値が設定されません。宣言した段階では、どこを指しているかわからないのです。値を設定しないうちにポインタを使ってしまうと、前項で解説したようにたいへん危険な結果になる場合があるので、注意してください。

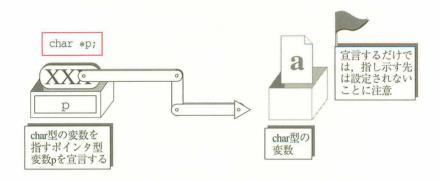



図6-29 ポインタ型変数の宣言

&演算子

［書式］&変数

ポインタを使いこなすには、**&演算子**と***演算子**という2つの演算子がカギになります。この2つの演算子の役割をしっかりと把握すれば、ポインタの機能を理解したといってもよいでしょう。

&演算子は、変数のアドレスを取り出します。次ページの図6-30のように、「&変数」という形で変数のアドレスを取り出し、それをポインタに代入することができます。図は、「char *型」、すなわち「charを指すポインタ型」変数pに、char型の変数cのアドレスをセットする様子を表しています。

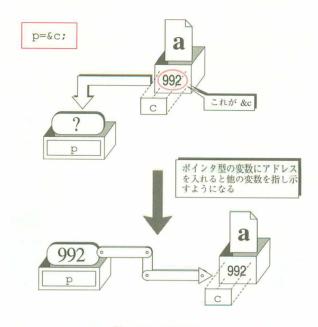



図 6-30　&演算子

アドレスを取り出すといっても、その値について意識する必要はまったくありません。&演算子で取りだしたアドレスをポインタに代入すると『ポインタがその変数を指し示すようになる』ことさえ把握していればよいのです。

*演算子

演算子は、ポインタの指し示している変数を操作するための演算子です。「」(アスタリスク)は掛け算の演算子でもありますが、ポインタの前に付くときにはポインタ演算子となります。

演算子は、いわば「分身の演算子」です。ポインタの前に「」を付けると、ポインタの指し示している変数の分身になります。たとえば、図 6-31 のように、「*p」はpの指し示している「変数c」の分身になりすまします。

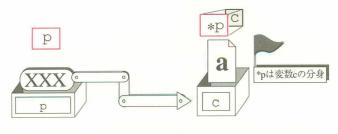



図 6-31　*演算子（その 1）

「＊p」は「変数 c」の分身ですから、あたかも「変数 c」であるかのように扱うことができます。「＊p」の値は変数 c の値であり、「＊p」に値を代入すれば、「変数 c」に代入することになります。図 6-32 のように、「変数 c」を使ってできる処理は、すべて「＊p」を使って置き換えることができるのです。

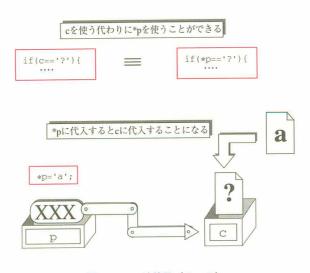



図 6-32　*演算子（その 2）

6.3.3　ポインタを使ったプログラミングの実際

ポインタをマスターするには、とにかく使ってみることがいちばんです。例題としていくつかのプログラムを作成してみましょう。プログラムの動作はそれほど難しいものではないので、ポインタの使い方に注目してください。

ポインタによる引数の受け渡し

次の図 6-33 は、前節で作成した例題プログラムの関数を使って日付の計算を行うためのプログラムを示しています。ltodate()関数（long to date の略）は、引数として西暦1年1月1日からの総日数を受け取り、年月日を求めて返します。

ltodate(西暦1年1月1日からの総日数，年へのポインタ，月へのポインタ，日へのポインタ)
総日数から年月日を求め，呼び出し側の変数に格納する

```
ltodate( long int days, int *y, int *m, int *d )
{
    for ( *y=1; days > ydays(*y,12,31) ; (*y)++ )
        days -= ydays(*y,12,31);
    for (*m=1; days > mdays(*y,*m) ; (*m)++ )
        days -= mdays(*y,*m);

    *d = days;
}


main()
{
    long int days3;
    int year, month, date;

    days3 = ldays(1964,3,28)+10000;
    ltodate(days3, &year, &month, &date );
    printf("私が生まれてから1万日目は %d年%d月%d日です。\n", year, month, date );
}
```


図 6-33　ltodate()関数

関数は、1つしか戻り値を返せませんから、3つの値を返すためにポインタを利用します。図 6-34 のように、ltodate()関数の引数として、main()関数内の変数を指し示すポインタを渡すのです。

```
ltodate( long int days, int *y, int *m, int *d )………総日数から年月日を求め、呼び出し側の変数に
{                                                      格納する
    for ( *y=1; days > ydays(*y,12,31) ; (*y)++ ) ……年を求める
        days -= ydays(*y,12,31);
    for (*m=1; days > mdays(*y,*m) ; (*m)++ ) …………月を求める
        days -= mdays(*y,*m);

    *d = days;……………………………………………………………日を求める
}


main()
{
    long int days3;
    int year, month, date;
                                 ……………………年月日を入れる変数へのアドレスを引数として渡す
    days3 = ldays(1964,3,28)+10000;      ……ltodate 関数が代入した値を表示する
    ltodate(days3, &year, &month, &date );
    printf("私が生まれてから1万日目は %d年%d月%d日です。\n", year, month, date );
}
```

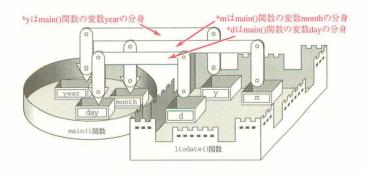



図 6-34　ポインタによる引数の受け渡し

main()関数では、&演算子を使って、変数 year、month、date のアドレスを ltodate()関数の引数として渡しています。

ltodate()関数では、引数変数として渡されるアドレスをポインタとして使用し、main()関数内の変数を処理しています。たとえば、「＊y」に代入すると、main()関数の変数 year に代入されます。このため、main()関数から見れば、ltodate()関数を呼び出して戻ってくるといつのまにか変数 year の値が変わっていることになります。

ここでひとつ注意することがあります。図6-35に示すように、「＊y＋＋」という式では、＊演算子よりも＋＋演算子の方が優先順位が高いので、「＊(y＋＋)」として扱われてしまいます。すると、ポインタの指し示している変数ではなく、ポインタそのものがインクリメントされてしまいます。

そこで、ポインタの指し示している変数をインクリメントするためには、「(＊y)＋＋」としなければなりません。

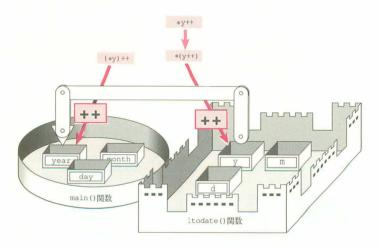



図6-35　「*演算子」と「++演算子」

この例題プログラムは、プログラム6-1のmain()関数以外の部分をそのまま利用しています。プログラムの実行例を、図6-36に示します。この例では、筆者の生まれた日からちょうど1万日目にあたる日を計算しています。みなさんも、自分の記念すべき日を計算してみてはいかがでしょうか。

```
私が生まれてから1万日目は 1991年8月14日です。
```

図6-36　実行例

ポインタと配列

ポインタを使って配列の処理を行うプログラムを作成します。ポインタを使わず、これまでの方法を使っても配列の処理は可能ですが、処理を高速化できることから、こういった処理にもポインタは非常によく利用されます。

図6-37に、4章108ページで作成したprtime_s()関数をポインタを使って処理するように書き直したプログラムを示します。プログラムの動作はおわかりでしょうから、理解しやすいと思います。ポインタによる処理方法に注目してください。

```
prtime_p(char *str,int t)
{
    char c;

    while(*str)  ……… ポインタの指す変数が0でない間繰り返す
        c=*str++; …… ポインタの指す変数の値をとり出し、cに代入する 同時にポインタをインクリメントする
        if(c=='?')
            printf("%2d:%02",t/100,t%100); }…… 文字が「?」だったら時刻を表示
        else
            putchar(c); ……… 「?」でなければ文字をそのまま表示
    }
}
```

図6-37　prtime_p()関数

例題プログラムのprtime_p()関数は、引数としてポインタ型変数strを受け取ります。strは、次の図6-38のように配列の先頭要素を指すように設定して呼び出すことにします。

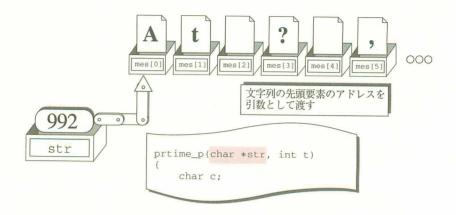



図6-38　prtime_p()関数の引数

ポインタのインクリメント

prtime_p()関数では、配列の先頭要素を「*str++」という式で取り出して変数cに代入すると同時に、ポインタであるstrの値をインクリメントしています（インクリメントは配列の先頭要素を取りだした後に実行されます）。ポインタをインクリメントすると、図6-39のように次の配列要素を指すようになります。

このプログラムでは、次に処理すべき配列要素のアドレスが常にポインタの値として保持されています。184ページの図6-23で解説したように、配列要素のアドレス計算が不要になるので、処理を高速化することができるのです。

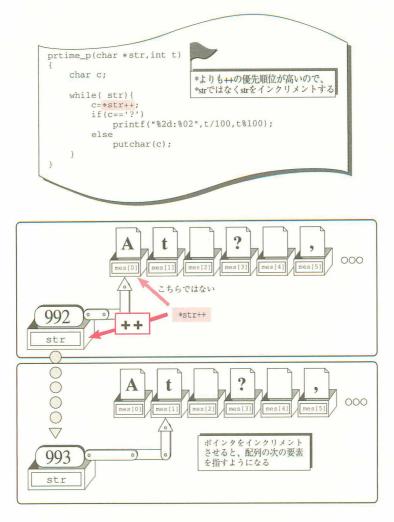



図 6-39　ポインタのインクリメント

配列名とアドレス

図 6-39 の上の部分は、例題プログラムの prtime_p()関数を呼び出す部分を示しています。図のように配列の先頭要素「mes [0]」のアドレスを「&mes [0]」という式で取り出しています。

配列の先頭要素のアドレスを取り出すには、もうひとつ方法があります。それは配列名を使う方法です。C言語では、図6-39のように配列名「mes」を変数名のように使うことができます。この場合、「mes」の値は配列の先頭要素のアドレスになります。

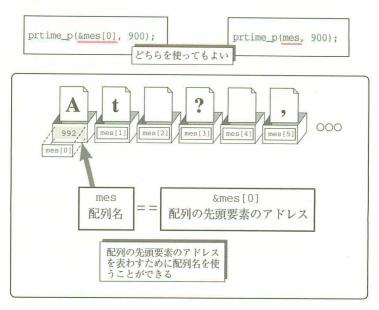



図6-40　配列名とアドレス

配列と引数

ここで4.2節で作成した文字列の例題プログラムのprtime_s()関数をもう一度思い出してください。この関数は、配列を引数として受け取っています。しかし、実をいうと配列ではなく、配列の先頭要素へのポインタを引数として受け取っているのです。

図6-41は、引数に配列を指定してprtime_s()関数を呼び出す例を示しています。前節で解説したように、配列名「mes」は、配列mesの先頭要素のアドレス「&mes [0]」を意味します。したがって、prtime_s()関数に渡されるのは、配列全体ではなく配列の先頭要素へのポインタなのです。

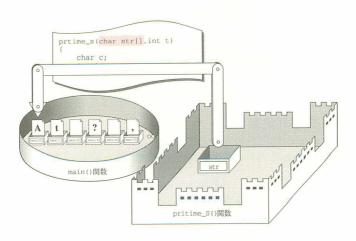



図 6-41　引数としての配列

ポインタと配列要素

配列名は配列の先頭要素のアドレスとして使えますが、逆にポインタをあたかも配列名のように扱うこともできます。つまり、図 6-42 に示すように、

```
prtime_s(char str[],int t)
{
    char c;
    int i;

    for(i=0; str[i]; i++){
        c=str[i];
        if(c=='?')
            printf
        else
            putcha
```

```
prtime_p2(char *str,int t)
{
    char c;
    int i;

    for(i=0; str[i]; i++){
        c=str[i];
        if(c=='0')
            printf("%2d:%02d",t/100,t%100);
        else
            putchar(c);
```

図 6-42　ポインタと配列要素

str［i］という形で配列要素を参照することができるのです。

結局、関数の引数の場合は、配列を宣言してもポインタを宣言したのとまったくかわりはありません。198ページで解説したように、配列を他の変数と同じように全部の値をコピーして関数に渡すのは、ムダが大きいからです。

ポインタと文字列定数

109ページの文字列定数のところで解説したように。char型の配列を引数として受け取る関数には、文字列定数を渡すことができます。配列を受け取るということは、結局ポインタを受け取ることと同じなので、prtime_s()関数を呼び出す場面でも、やはり文字列定数を使うことができます。

それはなぜかというと、文字列定数の値は、図6-43に示すように文字列の先頭要素のアドレスだからです。文字列定数を使用すると自動的に配列が用意され、その先頭要素のアドレスが値となるのです。このためアドレスを必要とする場面では、文字列定数を書くことができます。109ページで文字列定数は配列名と同じ扱いと説明したのは、実はこういう意味だったのです。

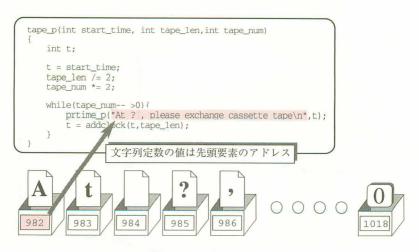



図6-43　文字列定数の値

ここで、ポインタにアドレスをセットする方法をまとめておきましょう。ポインタには、次の図6-44に示すような方法でアドレスをセットすることが

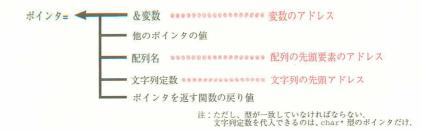



図 6-44　ポインタにセットできる値

できます。

int 型の配列とポインタ

int 型の配列とポインタの関係を解説して、ポインタの解説をひとまずしめくくります。4.2 節で作成した時刻表を検索するプログラムを、ポインタを使ったものに書き換えてみます。

このプログラムでは、時刻表を int 型の配列として表現しています。そこ

```
int main()
{
    int fromAtoB;
    int start;
    int t;
    int *ptr;
    fromAtoB = 236;
    start = 1000;

    for(ptr = tblA; ptr < tblA+6; ptr++)
        if(start < *ptr)
            break;
    if(ptr >= tblA+6){
        printf("A駅発の電車はもう終わりました.\n");
    } else{
        t =*ptr;
        printf("A駅を%2d:%02dに出て", t/100,t%100);
        t=addclock(t,fromAtoB);
        printf("B駅に%2d:%02dに着きます.\n",t/100,t%100);
    }
}
```

ptr が配列 tblA の先頭要素を指すようにする
ptr が配列中の要素を指している間繰り返す
ptr をインクリメントして次の要素を指すようにする
配列要素が start より大きければ繰り返して中断する

図 6-45　ポインタ版時刻表検索プログラム

で、int へのポインタを使ってプログラムを書き直すと、図 6-45 のようになります。

int 型の配列を指し示すポインタ型変数をインクリメントすると、char 型の配列を指し示す場合と同じように、やはり次の要素を指すようになります。

ところで、int 型の変数のメモリサイズは 2 バイトですから、int 型の配列の各要素は 2 バイトおきに並んでいることになります。ということは、int を指すポインタをインクリメントすると、図 6-46 のように 2 バイト分値が増えることになります。

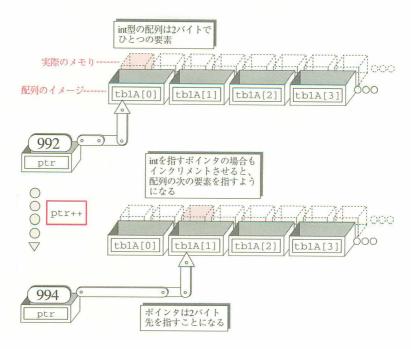



図 6-46　int を指すポインタのインクリメント

基本データ型の変数を＋＋演算子でインクリメントすると、変数の値を 1 増やしますが、ポインタの場合には、単に 1 増やすだけでは 1 バイト先のメモリを指すようになるだけで、次の要素を指し示すことにはなりません。

そこで、ポインタをインクリメントすると、そのポインタの指す型の変数

の占める大きさの分だけ値を増やして、次の要素を指すようにしているのです。ポインタをインクリメントするたびに、次の要素、その次の要素と、1つずつ先の要素を指すように変化します。同様にポインタをデクリメントすると、ひとつ前の要素を指すようになります。

ポインタへの加算

ここで、ポインタのインクリメントの応用として、ポインタへの足し算を考えてみましょう。

ポインタの値に足し算すると、加えた数だけ先の要素を指すようになります。たとえば、図6-47に示すようにポインタが配列のある要素を指しているとすると、これに2を加えた値は、2つ先の要素を指すアドレスになります。

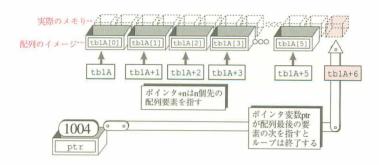



図6-47 ポインタへの加算

例題のプログラムでは、このことを利用してforループの終了判定を行っています。「tblA+6」という式の値は、図6-45に示したように、配列の最後の要素の次のアドレスとなります。ポインタ型変数がこの値に達したら、配列の大きさを超えてしまったことがわかります。

同様にポインタ型変数の値から引き算を行うと、その数だけ前の配列要素を指すアドレスとなります。

6.3.4　ポインタ利用上の注意

初期化されていないポインタ

ポインタを使う上で、十分注意しなければならないことがあります。それはポインタが「どこを指しているか」を常に把握していなければならないということです。ポインタは難解で、初心者はつまづきやすいとよく言われますが、多くの場合ポインタの指し示す場所をしっかり把握していないことが理解を妨げる原因です。このことに十分気をつけていれば、ポインタを使ったプログラムを理解することも決して難しくはありません。

こうしたことに気を配らずにプログラムを作成したために失敗してしまうケースをいくつか紹介しましょう。そのひとつは、ポインタにどこを指すのかという情報を入れないうちに、ポインタを「使って」しまうことです。

ポインタ型変数も他の変数と同じように、宣言しただけでは値は設定されません。つまり、でたらめの値が入っているので、どこを指しているかはわからないのです。たまたまどこかの変数を指していたりすると、ポインタによる代入はその変数をいつのまにか書き換えてしまうことになるでしょう。プログラムの一部やシステムの領域を指していたりすると、エラーや暴走の原因となってしまいます。

このような場合、UNIXなどのOSではメモリ保護機能が働き、エラーとしてプログラムが停止させられますが、MS-DOSなどのOSではメモリ保護機能がないのでシステムを破壊してしまうこともありえます。初期化していないポインタは、図6-48に示したように、いわば「爆弾」のようなものです。みなさんも爆弾を製造しないように十分気をつけてください。

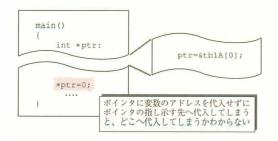
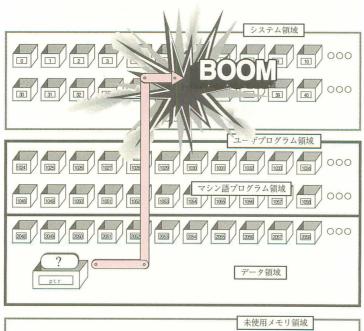
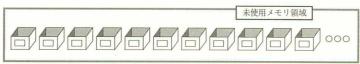



図 6-48　初期化されていないポインタ

その次によくやる失敗として、引数としてポインタを受け取る関数に、初期化されていないポインタを渡してしまう例があります。

ライブラリ関数を利用するためにマニュアルを調べる場合を考えてくださ

ltodate関数の呼び出し形式

ltodate(西暦1年1月1日からの総日数，年へのポインタ，月へのポインタ，日へのポインタ)
総日数から年月日を求め，呼び出し側の変数に格納する

（1）
```
main()
{
    int *yearptr, *monthptr, *dateptr;
    long days3;

    days3 = ldays(1964,3,28)+10000;
    ltodate(days3, yearptr, monthptr, dateptr);
    printf("私が生まれてから1万日目は%d年%d月%d日です。\n",
           *yearptr, *monthptr, *dateptr);
}
```
初期化していないポインタを使っている

（2）
```
main()
{
    int *yearptr, *monthptr, *dateptr;
    int year, month, date;
    long days3;

    days3 = ldays(1964,3,28)+10000;
    yearptr = &year;
    monthptr = &year;
    dateptr = &date;
    ltodate(days3, yearptr, monthptr, dateptr);
    printf("私が生まれてから1万日目は%d年%d月%d日です。\n",
           year, month, date);
}
```
ポインタを初期化すればＯＫ

（3）
```
main()
{
    int year, month, date;
    long days3;

    days3 = ldays(1964,3,28)+10000;
    ltodate(days3, &year, &month, &date);
    printf("私が生まれてから1万日目は%d年%d月%d日です。\n",
           year, month, date);
}
```
ポインタを渡す関数にはアドレスを渡してもよい

図6-49　ポインタを受け取る関数の呼び出し方

い。関数の呼び出し形式として、引数の型がポインタ型である場合はどうすればよいのでしょうか。

ここでは、194ページで作成したltodate()関数を例に解説しましょう。ltodate()関数の呼び出し形式は、図6-49のようにポインタ型の引数を渡すことになっており、そのポインタの指し示す変数に計算結果を代入します。

ポインタをよく理解していないと、図の(1)のようにポインタ型変数を用意して、それを引数として渡せばよいと考えがちです。ところが、このポインタは初期化されていないので、前の図6-48のような爆弾と化してしまい、結果を得ることもできません。

図の(2)のようにポインタにアドレスを代入して、変数を指し示すようにしておけば、その変数に結果が代入されます。いっそのこと、図の(3)のようにltodate()関数の引数としてアドレスを渡すのがいちばん簡単です。

ポインタ型の引数を受け取る関数は、言い換えればアドレスを受け取る関数です。ポインタを渡すということは、変数のアドレスを渡すということです。したがって、こうした場合には、わざわざポインタ型の変数を用意しなくても、アドレスを渡せばよいのです。

迷子のポインタ

もうひとつやってはいけない失敗に「迷子のポインタ」があります。やはりltodate()関数を例に解説しましょう。

ltodate()関数を次ページの 図6-50a のように定義するとどうなるでしょうか。

図のプログラムでは、配列に計算結果を格納し、その配列へのポインタを戻り値として返しています。この方法だと、一度に複数の値を戻り値として返すこともでき、便利そうです。

しかし、このプログラムには大きな問題点があります。4章で解説した「グローバル変数とローカル変数」のことを思い出してください。関数の中で宣言した変数はローカル変数であり、その関数の中でだけ有効です。変数に必要なメモリ領域は、関数に入った時点で確保され、関数の実行を終了するときに解放されます。

つまり、ltodate()関数を呼び出して得られたポインタの指す配列は、

int *ltodate(long days)	西暦1年1月1日からの総日数を年月日に変換し，int型の配列に入れて返す d[0]　…年を入れる d[1]　…月を入れる d[2]　…日を入れる (注：このプログラムは正しく動作しません！)

```
int *ltodate(long days)
{
    int d[3];

    for(d[0] = 1; days > ydays(d[0],12,31); d[[0]++)
        days -= ydays(d[0],12,31);

    for(d[1] = 1; days > ydays(d[0],d[1]); d[1]++)
        days -= mdays(d[0],d[1]);

    d[2] = days;
    return d;
}
```

図 6-50a　迷子のポインタ

ltodate()関数の実行を終了した時点で解放されてしまっているのです。解放されたメモリ領域は、別の関数を呼び出すとそのローカル変数領域として使われます。関数を呼び出さなくても、ハードウェア割り込み*4によって利用されることがあります。

このようなポインタは、まるで初期化していないポインタのように無効になってしまったと考えられます。いわば指し示す変数を見失った「迷子のポインタ」といえるでしょう。ローカル変数を指すポインタを関数の戻り値として返すことは、やってはいけないことなのです。

このような場合は、グローバル変数を用意してそのアドレスを返すか、あるいは、7章で解説するようにmalloc()関数などを利用して、ヒープ領域からメモリを確保し、そのアドレスを返すようにしなければなりません。

ポインタの基礎的な解説はここでひとまず締めくくります。7章でさらにポインタの応用例を詳しく紹介します。

＊4　周辺機器からの要求に応じて、現在の実行中のプログラムを一時中断して対応処理を実行する仕組みのこと。

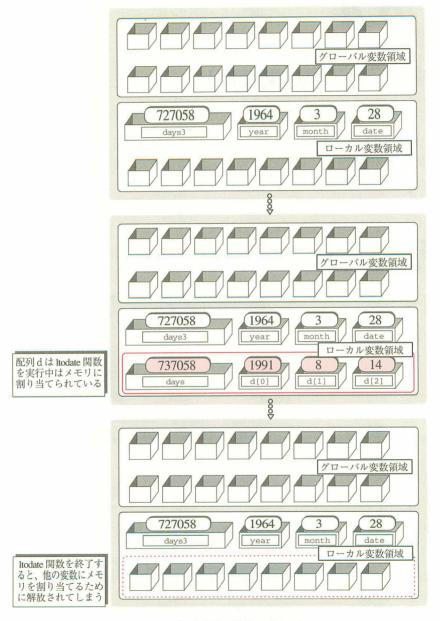



図 6-50b 迷子のポインタ

6.4

複合データ型(構造体)

複数の変数を組み合わせて作る複合データ型には、配列、構造体、共用体の3種類があります。共用体は、マシン語プログラムとのやりとりに使うなど、特殊な用途に使われることが多いので、本書では扱いません。配列については4章で詳しく解説しているので、本節では構造体を解説します。

構造体

構造体とは、互いに関連するいくつかのデータをまとめて1つの変数に入

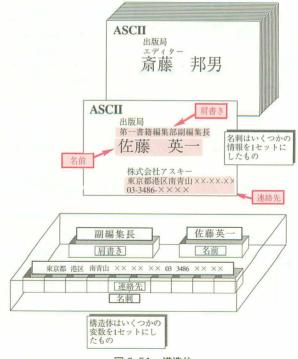



図6-51　構造体

れてしまうためのデータ型です。たとえば名刺を思い浮かべると、構造体の役割がよくわかります。図6-51を見てください。

名刺には、名前をはじめとして肩書きや連絡先といった情報が1枚の紙にまとめて書かれています。複数の情報を1セットとして扱っているわけです。これとまったく同じように、複数のデータ型をまとめて1つの変数として扱えるようにする機能が構造体です。

例題プログラムの構造体

本節では、構造体を使って時間を処理するプログラムを作成します。時間は、時と分という2つの情報から成り立っています。構造体を使うと図6-52のように時と分をワンセットにして、1つの時間という情報を表すことができます。

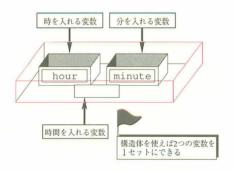



図6-52　例題プログラムの構造体

これまでの例題プログラムでは、時間を処理するために、100倍法という方法を考案して、時と分という2種類の数値を無理矢理1つの数値に押し込んで表現していました。しかし、構造体を使えば、自然に1つの変数として扱うことができます。

構造体の宣言

［書式］struct｛型名メンバ；…｝変数名；または struct タグ名｛型名メンバ；…｝

構造体型の変数も基本データ型の変数と同じように、宣言することによってメモリが確保されます。ただし、構造体の宣言は、次の図6-53のように2段階になります[*5]。

図6-53　構造体の宣言

構造体型の変数を宣言するには、struct という型名を使います。「struct」は structure（ストラクチャー）の略で、『構造』を意味します。

最初の宣言は、構造体型の**データ型の宣言**です。｛ ｝の中に1つの変数として扱いたいデータ型を、変数宣言と同じように並べて宣言します。そして次に、構造体型の**変数の宣言**をします。なお、ポインタ型変数のことを単にポインタと呼ぶように、構造体型の変数のことを単に構造体と呼ぶことがあります。

タグ

データ型の宣言は、いわば型紙を作る作業です。データ型の構造を型紙として用意し、その型紙を使って変数を作り出すのです。

次の図6-54のように、データ型の型紙に付けた名前のことを構造体の**タグ**と呼びます。タグには変数名と同じように自由に名前を付けることができ、一度型紙を作ってしまえば、後はそのタグを使っていくつでも構造体型変数を宣言することができます[*6]。

＊5　データ型の宣言と変数の宣言を同時に行うこともできますが、図のように2段階にした方がわかりやすいでしょう。

＊6　データ型の宣言と変数の宣言を同時に行う場合には、タグを付けない宣言も可能ですが、タグはかならず付けた方がよいでしょう。

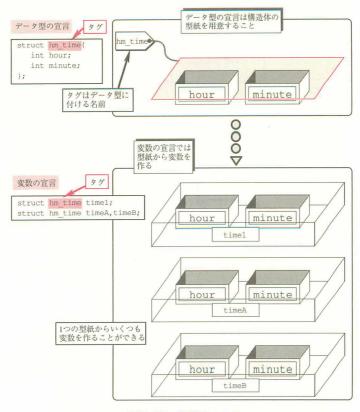



図 6-54　構造体のタグ

メンバ

［書式］ 構造体変数名.メンバ名　または　構造体へのポインタ->メンバ名

構造体中のひとつひとつのデータのことを、構造体の**メンバ**と呼びます。構造体型の変数は 1 つの変数として扱われますが、構造体どうしを加えるなど、まるごと演算の対象にすることはできません。加算などの演算は基本データ型でのみ可能です。

構造体を使って処理を行うには、内部の各メンバを個別に処理しなければなりません。構造体のメンバをアクセスするには、図 6-55 に示したように構造体変数名とメンバ名を「.」（ピリオド）でつなげた変数名を使います。

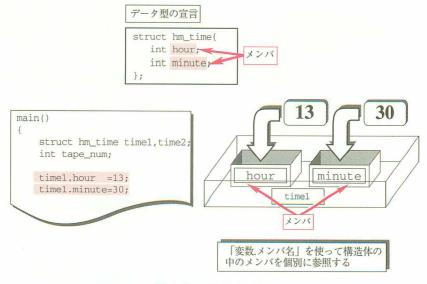



図 6-55　構造体のメンバ

構造体のメンバはひとつの変数として、他の変数とまったく同じように扱うことができます。たとえば、例題プログラムの構造体 time1 に時刻を設定するには、図 6-55 のように構造体のメンバひとつひとつに値を代入すればよいのです。

構造体と関数

次の図 6-56 は、4 章で作成したプログラムを構造体を使って書き直したものの一部です。

addclock_st()関数は、2 つの時間を引数として受け取り、それを加算した結果を戻り値として返します。構造体を使うことによって、関数の呼び出し方も、関数内の処理もわかりやすくなっていることに注目してください。100 倍法を使わなくても、自然に 2 つの情報を 1 つの変数として処理することができるからです。

構造体代入のコスト

実をいうと、図 6-56 のようなプログラムはあまり勧められる方法ではあり

```
struct hm_time {
        int hour;
        int minute;
};

struct hm_time addclock_st(struct hm_time time1,struct hm_time time2)
{
        struct hm_time time3;

        time3.hour=time1.hour + time2.hour;
        time3.minute=time1.minute + time2.minute;
        if (time3.minute >= 60) {
                time3.minute -= 60;
                time3.hour++;
        }
        return time3;
}

main()
{

        struct hm_time start,tape_len;
        int tape_num;

        start.hour=13;
        start.minute=30;
        tape_len.hour=0;
        tape_len.minute=23;
        tape_num=3*2;

        while (tape_num-- > 0) {
                start=addclock_st(start,tape_len);
                prtime_st("At ? ,Please exchenge cassete tape\n",start);
        }
}
```

構造体のデータ型の宣言は、関数の外で行ってもよい
構造体型で表された時間を受けとり加算した結果を構造体に入れて返す
時の加算
分の加算
分が 60 以上ならば
時に繰り上げる
13:30 開始
テープの長さは 23 分
テープの数は 3 本

図 6-56　時間加算プログラム

ません。なぜなら、次ページの図 6-57 に示すように、ムダな構造体コピーが何度も発生するからです。

関数の引数に変数を指定すると、変数の値が引数変数にコピーされます。また、関数の戻り値として構造体を返す場合もコピーが伴います。

ANSI 以前のC言語では、構造体のコピーはサポートされていませんでした。構造体を関数への引数として渡すことはもちろん、構造体を他の構造体に代入することもできなかったのです。これは、構造体のコピーはなるべく使わないようにと意識してC言語を設計したからです。

構造体は基本データ型と違って、内部にいくつもの変数を持っています。したがって、コピーするためには基本データ型の代入に比べて多くの実行時間やマシン語命令数が必要になります。

このような構造体のコピーは多くの場合ムダであり、構造体へのポインタを使った方が効率のよいプログラムを作成することができます。

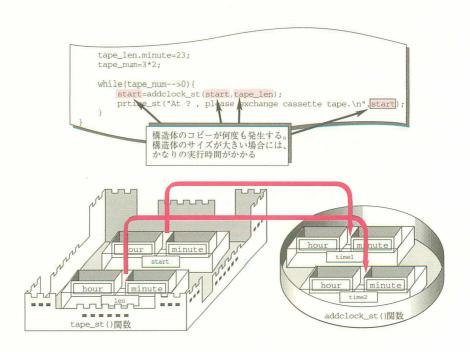



図 6-57　構造体のコピー

構造体へのポインタ

構造体を処理する関数を作成する場合には、引数として構造体そのものを渡すのではなく、なるべく構造体へのポインタを渡すようにしましょう。構造体へのポインタの宣言は、次の図6-58のように基本データ型へのポインタの宣言となんら変わりはありません。構造体のアドレスも&演算子を使って取り出すことができます。

図のように、関数に構造体へのポインタを渡すことにより、どんなにメモリサイズの大きな構造体であっても、簡単に関数に引数として引き渡すことができます。

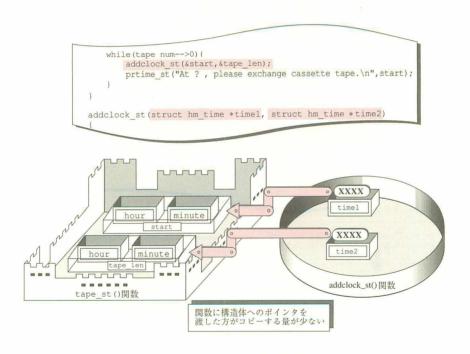



図6-58　構造体へのポインタ

-> 演算子

ところで、構造体へのポインタを使って構造体のメンバを処理する際には注意が必要です。なぜなら、構造体のメンバを取り出す演算子「.」の方が、ポインタによって変数を参照する演算子「*」よりも優先順位が高いからです。このため、図6-59のように、( )を使った面倒な記法を使わなければなりません。

構造体へのポインタに関しては、特別にメンバを参照する演算子が用意されています。図6-59のように「->」という演算子を使ってポインタの指している構造体のメンバを取り出すことができます。

こうして改良したaddclock_st()関数とプログラム全体を、図6-60に示します。なお、このプログラムの実行結果は、4章のプログラムと同じなので、実行例は省略します。

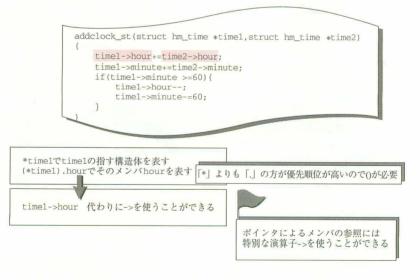



図6-59　->演算子

```
prtime_st(char *str, struct hm_time *t)……文字列中の'?'を時刻に置きかえて表示する
{                                      ……構造体型で表された時刻へのポインタを受けとる
    char c;

    while(*str) {
        c = *str++;
        if (c=='?')
            printf("%2d:%02d",t->hour,t->minute);
        else
            putchar(c);
    }
}
                   ……構造体型で表された時間へのポインタを受けとり加算した結果を1つ目の引数の指す
                                                                      構造体に入れて返す
addclock_st(struct hm_time *t1, struct hm_time *t2)
{
    t1->hour += t2->hour; ……………………………………時の加算
    t1->minute += t2->minute; …………………………分の加算
    if (t1->minute >= 60) { ………………………………分が60以上ならば
        t1->hour++; …………………………………………時に繰り上げる
        t1->minute -= 60;
    }
}

main()
{
    struct hm_time start,tape_len;
    int tape_num;

    start.hour   = 13;……………………………………………13:30 開始
    start.minute = 30;
    tape_len.hour   = 0;
    tape_len.minute = 23;…………………………………………テープの長さは23分
    tape_num = 3*2;…………………………………………………テープの数は3本

    while (tape_num-- >0) {
        prtime_st("At ? , please exchange cassette tape\n",&start);
        addclock_st( &start , &tape_len );
    }
}
```

図6-60 構造体を使ったテープ交換プログラム

〈本章で取り上げたプログラム〉 プログラム6-1

```
#include <stdio.h>

int isleapyear(int y)……………………………指定した年が閏年なら1を返し、閏年でなければ0を返す
{
    if ( y%4==0 && y%100!=0 || y%400==0 )……4で割った余りが0ならば4で割り切れることを利用する
        return 1;
    else
        return 0;
```

```
}

int mdays(int year, int month) ……指定した月の日数を返す
{
    int days;

    switch(month) { ……月によって処理を分岐させる
    case 1: case 3: case 5: case 7: ……大の月の日数は31日
    case 8: case 10: case 12:
        days=31;
        break;
    case 4: case 6: case 9: case 11: ……小の月の日数は30日
        days=30;
        break;
    case 2: ……2月は閏年でなければ28日
        days=28+isleapyear(year); ……閏年ならば29日
        break;
    default: ……1から12まで以外の数値が与えられた時の処理
        printf("mdays(%d,%d):parameter error\n",year,month);
        break;
    }
    return days;
}

int ydays(int year, int month, int date); ……1月1日からの総日数を返す
{
        int days,i;

        days=0;
        for (i=1 ; i<month ; i++)
                days+=mdays(year,i); ……1月から前月までの日数を合計する
        days += date; ……今月の日数を加える
        return days;
}

long int ldays(int y,int m,int d) ……西暦1年1月1日からの総日数を返す
{
    long int l; ……long int型の変数lを宣言する

    l=((long int)y-1)*365+(y-1)/4-(y-1)/100+(y-1)/400 + ydays(y,m,d);
                  ……365×昨年の西暦+昨年までの閏年の数+今年1月1日からの日数
    return l;
}

main()
{
    long int days1,days2;
    int year,month,date;
    days1 = ldays(2000,3,28); ……西暦1年1月1日から2000年3月28日までの総日数を求める
    days2 = ldays(1999,12,5); ……西暦1年1月1日から1999年12月5日までの総日数を求める
    printf("2000年3月28日は1999年12月5の%ld日後です。\n",days1-days2);

    days1 = ldays(2001,1,1); ……西暦1年1月1日から2001年1月1日までの総日数を求める
    days2 = ldays(1964,3,28); ……西暦1年1月1日から1964年3月28日までの総日数を求める
    printf("2001年1月1日は，私が生まれてから%ld日目です。\n",days1-days2);
}
        ……long int型の数値を表示するには%ldを使う(Appendix 2参照)
```

```
#include <stdio.h>
#include <ctype.h>

int addclock(int time1, int time2)
{                                  …… 100倍法で表した2つの時間を引数として受け取る
    int hour, minute;

    hour = time1/100 + time2/100;    …… 時の加算
    minute = time1%100 + time2%100;  …… 分の加算

    if (minute >= 60) {              …… 分が60以上であれば、時に繰り上げる
        hour++;
        minute -= 60;
    }

    return hour*100+minute;          …… 結果を100倍法で1つの数値にし、戻り値として返す
}

int subclock(int time1, int time2)
{                                  …… 100倍法で表した2つの時間を引数として受け取る
    int hour, minute;

    hour = time1/100 - time2/100;    …… 時の減算
    minute = time1%100 - time2%100;  …… 分の減算

    if (minute < 0) {                …… 分が0未満であれば、時を繰り下げる
        hour--;
        minute += 60;
    }

    return hour*100+minute;          …… 結果を100倍法で1つの数値にし、戻り値として返す
}

calc(int time,int num,char ope)      …… 演算記号に対応する計算を行う関数
{
    switch(ope) {
    case '+':
        time=addclock(time,num);
        break;
    case '-':
        time=subclock(time,num);
        break;
    default:
        time=num;
        break;
    }
    return time;
}

int main()
{
    char c;          …… 入力された文字を格納する変数
    char ope;        …… 1つ前の演算記号を記憶しておく変数
    int num;         …… 文字から数値への変換の途中結果を格納する変数
    int time;        …… 計算結果を格納しておく変数
    time=num=0;
    ope=' ';
```

```
        while ((c=getchar())!='\n') {
            if (isdigit(c))……入力された文字が数字かどうかを判定する関数
                num=num*10+(c-'0');……文字を数値に変換する
            else if (c=='+') {
                time=calc(time,num,ope);
                ope=c;
                num=0;
            } else if (c=='-') {
                time=calc(time,num,ope);
                ope=c;
                num=0;
            } else if (c=='=') {
                time=calc(time,num,ope);
                printf("%2d:%02d\n",time/100,time%100);
                time=num=ope=0;
            } else
                break;
        }
}
```

……1つ前の式を計算し、次の数値が入力された時に計算するための演算記号をとっておく

〈本章で取り上げたプログラム〉 プログラム6-3

```
#include <stdio.h>

int isleapyear(int y)……指定した年が閏年なら1を返し、閏年でなければ0を返す関数
{
    if ( y%4==0 && y%100!=0 || y%400==0 )……4で割った余りが0ならば4で割り切れることを利用する
        return 1;
    else
        return 0;
}

int mdays(int year, int month)……指定した月の日数を返す
{
    int days;

    switch(month) {……月によって処理を分岐させる
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        days=31;
        break;……大の月の日数は31日
    case 4: case 6: case 9: case 11:
        days=30;
        break;……小の月の日数は30日
    case 2:
        days=28+isleapyear(year);
        break;……2月は閏年でなければ28日 閏年ならば29日
    default:……1から12まで以外の数値が与えられた時の処理
        printf("mdays(%d,%d):parameter error\n",year,month);
        break;
    }
    return days;
}

int ydays(int year, int month, int date);……1月1日からの総日数を返す関数
```

```
{
        int days,i;

        days=0;
        for (i=1 ; i<month ; i++)
                days+=mdays(year,i); ……………………1月から前月までの日数を合計する
        days += date; ……………………今月の日数を加える
        return days;
}

long int ldays(int y,int m,int d) ……………………西暦1年1月1日からの総日数を返す関数
{
    long int l; ……………………long int 型の変数 l を宣言する

    l=((long int)y-1)*365+(y-1)/4-(y-1)/100+(y-1)/400 + ydays(y,m,d);
                                        ……365×昨年の西暦+昨年までのうるう年の数+今年1月1日
    return l;                             からの日数
}

ltodate( long int days, int *y, int *m, int *d )………総日数から年月日を求め、呼び出し側の変数に
{                                                       格納する関数
    for ( *y=1; days > ydays(*y,12,31) ; (*y)++ ) ……年を求める
        days -= ydays(*y,12,31);
    for (*m=1; days > mdays(*y,*m) ; (*m)++ ) ……………月を求める
        days -= mdays(*y,*m);

    *d = days; ……………………日を求める
}


main()
{
    long int days3;
    int year, month, date;
                                        ……………………年月日を入れる変数へのアドレスを引数として渡す
    days3 = ldays(1964,3,28)+10000;              ……ltodate 関数が代入した値を表示する
    ltodate(days3, &year, &month, &date );
    printf("私が生まれてから1万日目は %d年%d月%d日です。\n", year, month, date );
}
```

〈本章で取り上げたプログラム〉　プログラム6-4

```
#include <stdio.h>

main()
{
    int t;
    int tape_num;

    t = 1330; ……………………スタート時刻を 13:30 にする
    tape_num = 3;

    tape_p(t,46,tape_num);
}
```

```
tape_p(int start_time, int tape_len, int tape_num)
{
    int t;

    t = start_time;
    tape_len /= 2; ......................テープの長さは片面ごとなので tape_len÷2
    tape_num *= 2; ......................テープの数×2 が交換する回数

    while (tape_num-- >0) { .............テープの数が正の間くり返す
        prtime_p("At ? , please exchange cassette tape\n",t);
        t = addclock(t,tape_len);
    }
}


prtime_p(char *str,int t)
{
    char c;

    while (*str) { ......................ポインタの指す変数が 0 でない間繰り返す
        c = *str++; .....................ポインタの指す変数の値をとり出し、c に代入する
        if (c=='?')                      同時にポインタをインクリメントする
            printf("%2d:%02d",t/100,t%100); ...文字が「?」だったら時刻を表示
        else
            putchar(c); .................「?」でなければ 文字をそのまま表示
    }
}
int addclock(int time1, int time2) ......100 倍法で表した 2 つの時間を引数として受け取る
{
    int hour, minute;

    hour = time1/100 + time2/100; .......時の加算
    minute = time1%100 + time2%100; .....分の加算

    if (minute >= 60) { .................分が 60 以上であれば、時に繰り上げる
        hour++;
        minute -= 60;
    }

    return hour*100+minute; .............結果を 100 倍法で 1 つの数値にし、戻り値として
}                                        返す
```

〈本章で取り上げたプログラム〉　プログラム6-5

```
#include <stdio.h>                       136 ページの配列版プログラムと比べてみてください

int tblA[] = { 930,1052,1205,1319,1427,1610 };

main()
{
    int fromAtoB;
    int start;
    int i,t;
    int *ptr;
```

```
    fromAtoB = 236;                                  ptr が配列 tblA の先頭要素を指すようにする
    start = 1000;                                    ptr が配列中の要素を指している間繰り返す
                                                     ptr をインクリメントして次の要素を指すようにする
    for ( ptr=tblA ; ptr < tblA+6 ; ptr++ )
        if (start < *ptr)                            配列要素が start より大きければ繰り返して中断する
            break;
    if (ptr >= tblA+6 ) {
        printf("A 駅発の電車はもう終わりました.\n");
    } else {
        t = *ptr;
        printf("A 駅を %2d:%02d に出て ", t/100,t%100);
        t = addclock(t,fromAtoB);
        printf("B 駅に %2d:%02d に着きます.\n",t/100,t%100);
    }
}

int addclock(int time1, int time2)                   100 倍法で表した 2 つの時間を引数として受け取る
{
    int hour, minute;

    hour = time1/100 + time2/100;                    時の加算
    minute = time1%100 + time2%100;                  分の加算

    if (minute >= 60) {                              分が 60 以上であれば、時に繰り上げる
        hour++;
        minute -= 60;
    }

    return hour*100+minute;                          結果を100 倍法で 1 つの数値にし、戻り値として返す
}
```

〈本章で取り上げたプログラム〉 プログラム6-6

```
#include <stdio.h>

struct hm_time {
    int hour;
    int minute;
};

prtime_st(char *str, struct hm_time *t)              文字列中の'?'を時刻に置きかえて 表示する関数
{                                                    構造体型で表された時刻へのポインタを受けとる
    char c;

    while(*str) {
        c = *str++;
        if (c=='?')
            printf("%2d:%02d",t->hour,t->minute);
        else
            putchar(c);
    }
}
                 構造体型で表された時間へのポインタを受けとり、加算した結果を 1 つ目の引数の指す構造体に入れて返す
addclock_st(struct hm_time *t1, struct hm_time *t2)
{
```

```
    t1->hour += t2->hour; ·························時の加算
    t1->minute += t2->minute; ·····················分の加算
    if (t1->minute >= 60) { ·······················分が 60 以上ならば
        t1->hour++;
        t1->minute -= 60;       ···················時に繰り上げる
    }
}

main()
{
    struct hm_time start,tape_len;
    int tape_num;

    start.hour   = 13;·····························13:30 開始
    start.minute = 30;
    tape_len.hour   = 0;
    tape_len.minute = 23;··························テープの長さは 23 分
    tape_num = 3*2;································テープの数は 3 本

    while (tape_num-- >0) {
        prtime_st("At ? , please exchange cassette tape\n",&start);
        addclock_st( &start , &tape_len );
    }
}
```

第7章

C言語をとりまく世界

本書もいよいよ終わりに近づいてきました。ここまで本書を読んだみなさんは、すでにC言語をマスターしたといってもよいでしょう。プログラムを作成するためのカギとなる部分はすべて解説しました。

しかし、実際にはプログラミング言語を知っているだけでプログラムを作れるわけではありません。プログラムを作成するための、プログラミング環境をよく理解する必要があります。

今後は、C言語をとりまくプログラミング環境として、次のようなことをそれぞれ学習していくとよいでしょう。

- プログラムの開発環境
- アルゴリズムとデータ構造
- プログラムの実行環境

C言語は、言語をとりまくこうした環境との連携プレーがうまくいくように設計されています。本書の最終章である本章は、これらのテーマについてそれぞれ基礎的なことを解説して、締めくくります。今後の学習の指針として大いに役立ててください。

本章で解説する項目

	データ型	部品化機能	制御構造
7.1		分割コンパイルとリンク	
7.2	アルゴリズムとデータ構造 リスト構造		
7.3		プログラムの実行環境 オペレーティングシステム ファイル入出力 コマンドパラメータ	

7.1

プログラムの開発環境

分割コンパイルとリンク

C言語では、「関数」という形でプログラムを部品化します。これまでの章で示してきた例題プログラムでも、同じ関数を何度も部品として再利用してきました。本節では、プログラム部品の活用方法をより詳しく解説します。

プログラム部品の再利用のカギは**分割コンパイル**と**リンク**です。1章の23ページで、ソースプログラムから実行可能プログラムを作成する手順を解説しましたが、この手順をさらに詳しく図解すると図7-1のようになります。

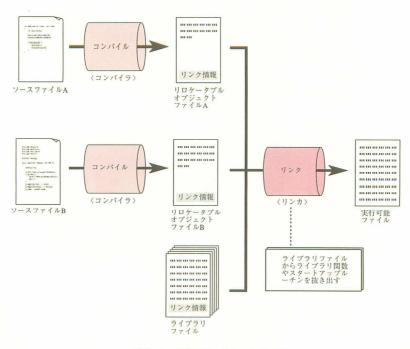



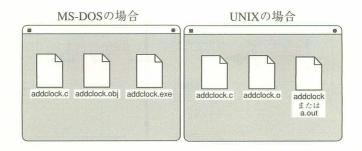
図7-1　分割コンパイルとリンク

図のように、ソースプログラムをもっと小さな複数のソースファイルに分割しておき、各ファイルごとにコンパイルを行うことができます。リンクは、コンパイルによって作成されたオブジェクトファイルを結合して実行可能ファイルを作成する作業です。

コンパイルによって作成されるオブジェクトファイルは、**リロケータブルオブジェクト**と呼ばれ、関数やグローバル変数の名前やそのアドレスなどの**リンク情報**を含んだ特殊なファイル形式になっています。リンクでは、このリンク情報をたよりにマシン語プログラムどうしを結合する作業を行います。このとき、ライブラリ関数やスタートアッブルーチン[*1]など、必要なプログラム部品を自動的に抜き出す作業も行います。

オペレーティングシステムによるファイル名の違い

MS-DOSの場合、ソースファイル「addclock.c」をコンパイルすると、リロケータブルオブジェクト「addclock.obj」が作成されます。他のオブジェクトファイルやライブラリをリンクすると、実行可能ファイル「addclock.exe」が作成されます。

UNIXでは、オブジェクトファイルが「addclock.o」、実行可能ファイルが「addclock」あるいは「a.out」となります。

みなさんも、コンパイルとリンクの作業を行って、このようなファイルが作成されることを確認してみてください。

＊1　スタートアップルーチンとは、各種初期設定や後で解説するコマンドライン文字列をmain()関数への引数に変換するプログラムのことです。

プログラムの部品化

分割コンパイルとリンクの機能によって、図7-2のようにプログラム部品の再利用が可能になります。たとえば、図7-2に示したソースファイルBは、プログラムAの部品としても利用され、プログラムCの部品としても利用されています。このように分割したプログラム部品の1つ1つのことをモジュールと呼ぶことがあります。

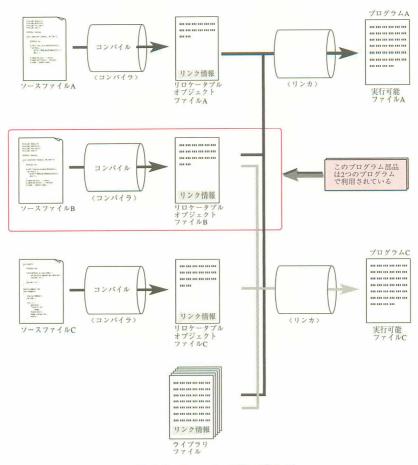



図7-2 プログラム部品の再利用

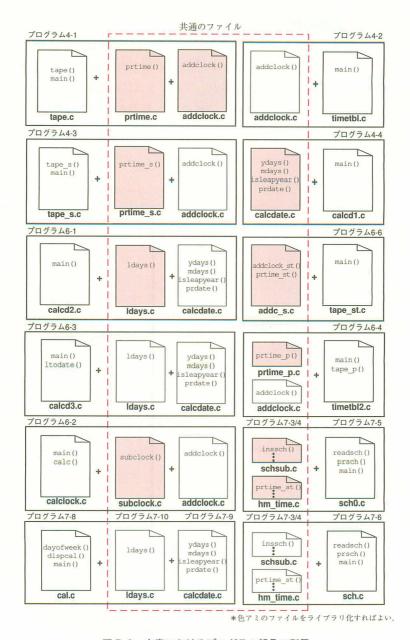



図 7-3　本書におけるプログラム部品の利用

本書の例題プログラムも、分割コンパイルとリンクによるプログラム部品として再利用が可能です。実際の手順を説明しましょう。

まず、これまでに作成したプログラムは、図7-3のように関数群ごとにソースファイルに分割してください。各ソースファイルは、図のように再利用可能な関数を別ファイルとして作成します。そして、実行可能プログラムごとに固有な部分は、main()関数やいくつかの関数だけにします。

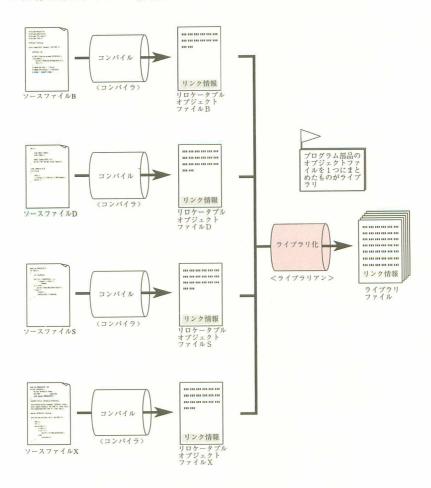



図7-4 ライブラリ

実行可能ファイルを作成するには、それぞれ独立にコンパイルしてリンクします。

ライブラリ

図7-1にあるライブラリファイルについて、解説しましょう。

ライブラリファイルとは、前ページの図7-4のようにプログラム部品としてのソースファイルをコンパイルしてできたオブジェクトファイルを、リンク情報を保ったまま1つのファイルにまとめたものです。

C言語処理系の標準ライブラリ関数は、ライブラリファイルとして提供されており、プログラム中で呼び出されるライブラリ関数は、リンク時に自動的にライブラリファイルから抜き出されて、実行可能ファイルに連結されます。

私たちが作成するプログラム部品も、ライブラリアンというツールを使ってライブラリにしておくことができます。

プログラム部品の数が多くなってくると、オブジェクトファイルをいちいち指定するのが面倒になってきます。そこで、ライブラリファイルにしておけば、リンク時にいちいちオブジェクトファイルを指定しなくても、リンカがライブラリファイルの中から自動的に必要なオブジェクトファイルを抜き出してくれるようになるのです。

プログラム部品がある程度たまってきたら、ライブラリアンを使ってライブラリファイルを作成するとよいでしょう。

関数プロトタイプ宣言

［書式］ 型名 関数名(［型名 ［引数］ ［,型名 ［引数］ ...］］);

C言語のプログラム部品化機能には弱点があります。それは、プログラム部品である関数の呼び出し方法を間違えても、文法的なエラーとしてチェックされないことです。たとえば、コンパイル作業はファイルごとに独立に行われますから、他のファイルにある関数を呼び出す際に、戻り値の型や引数の数や型を間違えてもそれをチェックすることはできません。

関数プロトタイプ宣言は、そうした弱点を補うため、ANSI-C規格に導入

された機能です。関数の情報をあらかじめ宣言しておき、コンパイラに関数の呼び出し方をチェックしてもらうのです。関数プロトタイプ宣言は、関数定義の先頭部分を抜きだしたものといってもよいでしょう。図7-5は、addclock()関数のプロトタイプ宣言部分を示しています。このように、プロトタイプ宣言をしておけば、addclock()関数の呼び出しは、かならずコンパイル時にチェックが行われるようになります。

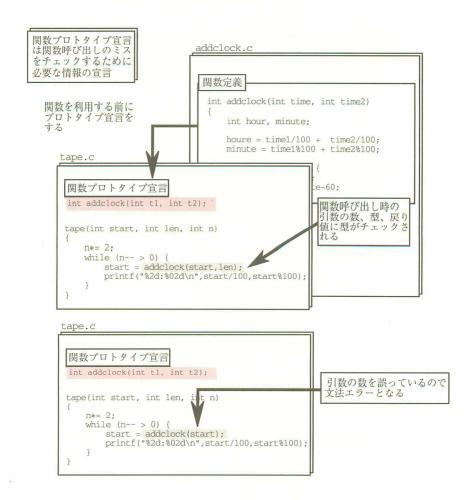



図7-5　関数プロトタイプ宣言

なお、図では引数変数の名前まで抜き出していますが、チェックには型名しか利用されないので引数変数の名前は省略してもかまいません。

あらかじめ宣言をしていない関数は、int 型を返す関数として処理されますから、図 7-6 のような場合には、コンパイルエラーとなってしまいます。したがって、int 型以外の型を返す関数を利用する際には、関数を呼び出す前に、かならず宣言しなければなりません。

ldays.c

```
long int ldays(int y, int m, int d)
{
    long int l;

    l = ((long int) y-1)*365+ (y-1)/4-(y-1)/100+ (y-1)/400 + ydays(y, m, d);

    return l;
}
```

calcd.c

int型以外の戻り値を返す関数は必ず宣言が必要

```
main()
{
    long int days1, days2;
    long int ldays(int y, int m, int d);

    days1 = ldays(2000, 3, 28);
    days2 = ldays(1999, 12, 5);
    printf("2000年3月28日は1999年12月5日の%ld日後です。 \n", days1-days2);

    days1 = ldays(2001, 1, 1);
    days2 = ldays(1964, 3, 28);
    printf("2001年1月1日は、私が生まれてから%ld日後です。 \n", days1-days2);
}
```

図 7-6　関数の型宣言の必要な場合

実は、これには例外があり、図 7-7 のように同じソースファイル内で、関数呼び出しよりも前に関数が定義されている場合には、関数が宣言された場合と同様に処理されます。6 章の例題プログラムでは、本来は関数プロトタイプを宣言すべきなのですが、説明を避けるため敢えて省略していたのです。

```
long int ldays(int y, int m, int d)
{
    long int l;

    l = ((long int) y-1)*365+ (y-1)/4-(y-1)/100+ (y-1)/400 + ydays(y, m, d);

    return l;
}

main()
{
    long int days1, days2;

    days1 = ldays(2000, 3, 28);
    days2 = ldays(1999, 12, 5);
    printf("2000年3月28日は1999年12月5日の%ld日後です。 \n", days1-days2);

    days1 = ldays(2001, 1, 1);
    days2 = ldays(1964, 3, 28);
    printf("2001年1月1日は、私が生まれてから%ld日後です。 \n", days1-days2);
}
```

同じソースファイル内で先に定義されていれば、宣言されたことになる

```
main()
{
    long int days1, days2;
    long int ldays(int y, int m, int d);

    days1 = ldays(2000, 3, 28);
    days2 = ldays(1999, 12, 5);
    printf("2000年3月28日は1999年12月5日の%ld日後です。 \n", days1-days2);
    days1 = ldays(2001, 1, 1);
    days2 = ldays(1964, 3, 28);
    printf("2001年1月1日は、私が生まれてから%ld日後です。 \n", days1-days2);
}

long int ldays(int y, int m, int d)
{
    long int l;

    l = ((long int) y-1)*365+ (y-1)/4-(y-1)/100+ (y-1)/400 + ydays(y, m, d);

    return l;
}
```

同じソースファイル内でも定義する前に呼び出すためには宣言が必要

図 7-7 関数定義と関数プロトタイプ宣言

しかし、関数プロトタイプ宣言は省略せずに、かならず宣言するようにしましょう。そうすれば、プログラムを複数のファイルに分割した場合に、プログラムミスの防止になります。

関数プロトタイプ宣言は面倒に思えるかもしれませんが、後で解説するヘッダファイルのインクルード機能を利用すれば、それほどの手間ではありません。

関数宣言

ANSI以前のC言語には、関数プロトタイプ宣言の機能がありませんでした。その代わり図7-8のように、関数の返す型だけを関数宣言していました。

```
main()
{
    long int days1, days2;
    long int ldays();
    days1 = ldays(2000, 3, 28);
    days2 = ldays(1999, 12, 5);
    printf("2000年3月28日は1999年12月5日の%ld日後です。 \n", days1-days2);

    days1 = ldays(2001, 1, 1);
    days2 = ldays(1964, 3, 28);
    printf("2001年1月1日は、私が生まれてから%ld日後です。 \n", days1-days2);
}

long int ldays(y, m, d)
int y;
int m;
int d;
{
    long int l;

    l = ((long int) y-1) * 365 + (y-1)/4 - (y-1)/100 + (y-1)/400 + ydays(y, m, d);

    return l;
}
```


図7-8　関数の型宣言

ANSI-C でも、ANSI 以前のプログラムとの互換性を保つため、このような関数宣言を行うことができます。ただし、関数宣言によって関数の返す型は正しくチェックされますが、関数の引数に関してはノーチェックとなってしまいます。

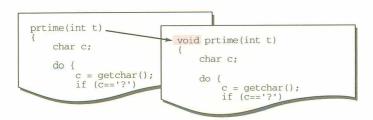
include

［書式］ #include 〈ファイル名〉　または　#include "ファイル名"

関数プロトタイプ宣言は、237 ページの図 7-5 のように呼び出す関数内で宣言してもよいのですが、もっとよい方法があります。それは**ヘッダファイ**

void 型

これまでの例題プログラムでは、戻り値を返さない関数の型は宣言していませんでしたので、関数の型 int が省略されたと解釈されます。ところが、実際には値を返さないにもかかわらず int 型を返すように扱われると、誤って関数の戻り値を変数に代入するなどしてもエラーとしてチェックされず、プログラムの誤動作の元になります。

そこで、値を返さない関数は、図のように「void 型」として宣言します。void は『空の』という意味で、何も値を返さないことを表します。

また、引数を持たない関数のプロトタイプ宣言では関数宣言と区別するため引数の型を void 型として宣言します。

みなさんも、これまでの例題プログラムを複数ソースファイルに分割する際に、値を返さない関数の型を void 型に変更してください。

ルに関数プロトタイプを書いておき、それを**インクルード**して使う方法です。

ヘッダファイルのヘッダとは『頭書き』という意味で、関数プロトタイプ宣言や構造体のデータ型宣言などをまとめておくファイルです。ヘッダファイルには「.h」で終わるファイル名を付けます。また、インクルードとは『包含する』という意味で、ソースファイルの中に他のソースファイルを取り込むための仕組みです。図7-9は、インクルードの仕組みを表したものです。

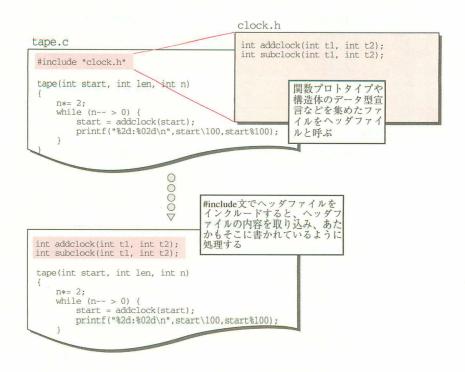



図7-9　ヘッダファイルをインクルードする

これまでの例題プログラムでも、「# include <stdio.h>」という命令を使っていました。これは、ライブラリ関数printfやgetcharのプロトタイプ宣言を取り込むためです。# includeはインクルードのための命令で、stdio.h（stdioはstandard i/oの略）はヘッダファイルの名前です。上の図7-9のように#include命令を使うと、あたかもそこにヘッダファイルの中身が書かれてい

るかのように処理します。

なお、# include 命令に関しては自由なスタイルで書くことはできず、かならず図 7-9 ように行頭に書かなければなりません。

ライブラリ関数のプロトタイプ宣言は、一連のヘッダファイル群としてC言語処理系の一部として提供されています。ライブラリ関数を使う場合は、マニュアルなどを参照して、関数に対応するヘッダファイルをかならずインクルードしましょう*2。

ここで、本書の例題プログラム 6-3 を部品化して作成した calcdate.c のヘッダファイルを、図 7-10 に示します。これまでの例題プログラムも、このヘッダファイルをインクルードするように修正してください。

```
int isleapyear(int year);
int mdays(int year, int month);
int ydays(int year, int month);
long int ldays(int year,int month,int day);
```

図 7-10　本章のプログラム 6-3 のヘッダファイル

図7-9では、ヘッダファイルの名前を「" "」で囲んでいます。これに対し、stdio.h は「< >」で囲んでいました。この違いは、ヘッダファイルを置く格納場所の違いを意味しています。「" "」で囲むと、ソースファイルと同じディレクトリからヘッダファイルを探してインクルードします。これに対し、「< >」で囲むと、ライブラリ関数用のヘッダファイルを置いてあるディレクトリからヘッダファイルを探してインクルードします*3。したがって、ライブラリ関数のヘッダは、<stdio.h> のように「< >」で囲み、自作の関数の場合は、"calcdate.h"のように「" "」で囲みます。

＊2　Appendix 2 にライブラリ関数に関するリファレンスの読み方と使い方を示しましたので、参考にしてください。

＊3　正確には、「" "」ではソースファイルと同じディレクトリおよび標準ヘッダのディレクトリの両方からヘッダファイルを検索するのに対し、「< >」では標準ヘッダのディレクトリしかヘッダファイルを検索しません。

define

#で始まるもう一つの命令、**マクロ定義**を紹介しましょう。図7-11に、マクロ定義の仕組みを示します。

```
#define TABLESIZE 6        マクロ定義

int main()
{
    int fromAtoB;
    int start;
    int i,t;

    fromAtoB = 236
    start = 1000;

    for (i=0; i < TABLESIZE
        if (start < tblA[i])
            break;
    if (i >= TABLESIZE ) {
        printf("A駅発の電車はも
    } else {
        t = tblA[i];
        printf("A駅を%2d:%02dに出
        t = addclock(t, fromAtoB)
        printf("B駅に%2d:%02dに着
    }
}
```

マクロ展開

マクロ展開

```
int main()
{
    int fromAtoB;
    int start;
    int i,t;

    fromAtoB = 236
    start = 1000;

    for (i=0; i < 6 ; i++)
        if (start < tblA[i])
            break;
    if (i >= 6 ) {
        printf("A駅発の電車はもう終りました。\n");
    } else {
        t = tblA[i];
        printf("A駅を%2d:%02dに出て ", t/100, t%100);
        t = addclock(t, fromAtoB);
        printf("B駅に%2d:%02dに着きます\n", t/100, t%100);
    }
}
```

図7-11　マクロ定義

#define（ディファイン）は、マクロ名を定義する命令です。図7-11を見てください。プログラム中でマクロ名を使うと、マクロ定義で割り当てた文字列に置き換えられます。このことを**マクロ展開**と呼びます。

この図から、マクロ定義の持つ大きなメリットがわかっていただけたでしょうか。たとえば、プログラム中の配列の要素数6を10に変更することを考えてください。マクロ定義を行っていれば、定義部分だけを変更するだけで、他の部分は自動的に変更が反映されます。ところが、マクロ定義を行わなければ、プログラム中で数値6を指定している部分をすべて自分の手で10に変更しなければなりません。このように、マクロ定義は、プログラムの修正をたいへん容易にするのです。

ただ、#define命令も、#include命令と同様に、自由なスタイルでは書け

ず、行頭に書かなければなりません。また、その行の行末までマクロ定義として扱われます。

なお、引数を持つマクロ名を定義することもできますが*4、最初は単純なマクロ定義だけを利用する方がよいでしょう。気付きにくい落とし穴があるなど、プログラミングに慣れるまで利用方法が難しいからです。

プリプロセッサ

includeや# defineは1行に1つの命令しか書くことができず、またその行に他の命令を書くことはできません。C言語ではプログラムのスタイルは自由なはずなのに、おかしいと思う人もいるでしょう。

実をいうと、#ではじまる命令は、本来はC言語の一部ではなく**プリプロセッサ命令**と呼ばれる命令です。コンパイル作業では、図7-12のようにプリプロセッサ命令を処理するプリプロセス処理と、C言語プログラムをマシン語プログラムに変換するコンパイル処理の2段階の処理を行っているのです。

プリプロセスを行うプリプロセッサは、コンパイル作業の中で自動的に呼び出されるので、とくに意識する必要はありません。ただし、C言語の文法とは無関係に、ファイルの挿入やマクロの置換が行われることを理解しておいてください。

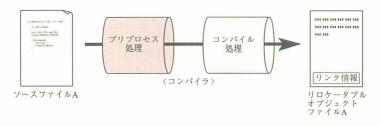



図7-12　プリプロセッサ

*4　実はgetchar()やputchar()は関数ではなく、こうしたマクロ定義されています。

typedef

構造体型の変数や引数の宣言はだらだらと長くなりがちで、入力が面倒なだけでなく、プログラムの読みやすさも損なわれます。そこで、typedef による型定義機能を利用することにしましょう。

typedef 宣言は図 7-13 のように、「typedef」の後に変数宣言と同じ書式で新しい型名を定義します。typedef で定義した新しい型名は、C 言語にあらかじめ備わっている型であるかのように使用することができます。

図の例では、int 型と同じ型の number 型を定義しています。以後、number 型の変数として変数宣言を行い、int 型の変数と同じように使用することができます。

typedef 文をヘッダファイルに定義しておくと、プログラムに改良を加えて number 型を long 型に変更することになった場合でも、ヘッダファイルの typedef 文の部分だけの変更で済みます。

また、時間を格納する構造体 struct　hm_time 型も、図のように HM_TIME 型として再定義することで、簡潔に宣言することができるようになります。

なんだか、新しいデータ型を作ったような気になりませんか？

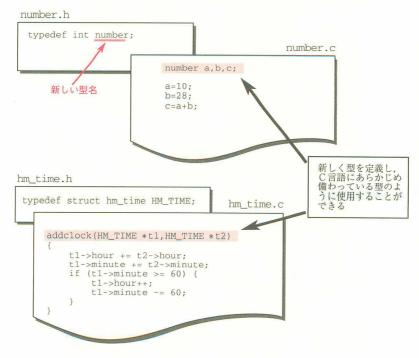



図7-13　typedef

extern

分割コンパイルを利用する際には、グローバル変数の扱いに工夫が必要になります。

グローバル変数は関数の外で宣言し、どの関数からもアクセスできる変数のことですが、そのままでは他のソースファイルで宣言したグローバル変数をアクセスすることはできません。

他のソースファイルのグローバル変数をアクセスするためには、**extern 宣言**を使います。次ページの図7-14のようにexternを付けてグローバル変数を宣言すると、その変数は他のソースファイルで宣言されているものとして扱われます。

extern宣言によるグローバル変数は、コンパイル作業時には仮のアドレスが割り当てられます。そして、リンク時にグローバル変数の実体を持つオブジェクトファイルと結合する際に、あらためてアドレスが決定されます。

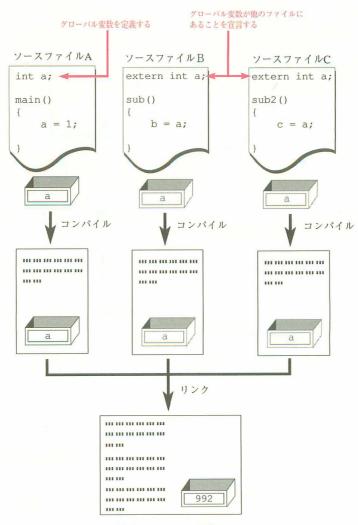



図7-14　extern 宣言

7.2

アルゴリズムとデータ構造

プログラム例をいくつも読むうちに、また自分でいくつかプログラムを書いていくうちに、典型的な処理についてのテクニックが次第にわかってきます。そうして、いわば「プログラミングの慣用句」を身につけていくと、今度はそれを応用して独自のプログラミングができるようになります。そういった意味でも、もし処理方法が思い浮かばないときは、プログラム集などから似たような処理を探して真似してみるとよいでしょう。

プログラミングが得意な人というのは、情報を表現するための変数の組み合わせ方（データ構造）や、プログラムの処理手順の組み立て方（アルゴリズム）を豊富に身につけた人のことです。みなさんも、アルゴリズムとデータ構造を盗み取るつもりでプログラム例を読んでください。

本節では、リスト構造と呼ばれるデータ構造と、リスト構造を処理するアルゴリズムを紹介します。

リスト構造

一口にいって、リスト構造とは、図7-15のように一連のデータが矢印によってつながっているデータ構造のことです（リスト構造については、6章の184ページで解説しています）。図のように、途中のつながりを容易に組み換えられることがリスト構造の特徴です。

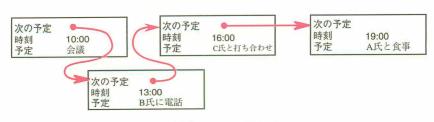



図7-15　リスト構造

リスト構造は、構造体とポインタの組み合わせで表現することができます。図 7-15 のようなスケジュール表を表すデータ型は、図 7-16 のように定義します。

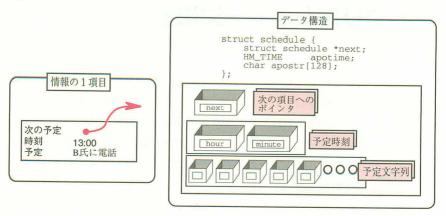



図 7-16　リスト構造のデータ型

スケジュール処理プログラム

早速、このデータ型を使って、スケジュール情報を処理する例題プログラムを作成してみましょう。このプログラムは、後で改良することを考えて 3 つモジュールと 2 つのヘッダファイルに分割しておきます。

```
/*
    スケジュール管理プログラム用ヘッダファイル
*/

#define MAXAPOSTR 128
struct schedule {
    struct schedule *next;
    HM_TIME     apotime;
    char apostr[MAXAPOSTR];
};
typedef struct schedule SCHEDULE;

void delsch(SCHEDULE *presch, SCHEDULE *sch);
void inssch( SCHEDULE *presch , HM_TIME *t , char *str );
void searchsch( HM_TIME *t , char *str );

extern SCHEDULE *schtop;
```

（注記：struct schedule の宣言 …… スケジュールの一項目のデータ型の宣言／typedef …… 型名の置き換え／void 関数群 …… スケジュール管理関数のプロトタイプ宣言／extern …… グローバル変数の宣言）

図 7-17　スケジュール管理プログラムのヘッダファイル「sch.h」

```
/*
   時間構造体定義ヘッダ
*/

struct hm_time {
    int hour;
    int minute;
};
typedef struct hm_time HM_TIME;

void prtime_st(char *str, HM_TIME *t);
int cmpclock(HM_TIME *t1, HM_TIME *t2);
void break_time(HM_TIME *t,int hm);
int bind_time(HM_TIME *t);
```


図 7-18　スケジュール管理プログラムのヘッダファイル「hm_time.h」

```
#include <stdio.h>
#include "hm_time.h"

void prtime_st(char *str, HM_TIME *t)
{
    char c;
    int i;

    while(*str) {
        c = *str++;
        if (c=='?')
            printf("%2d:%02d",t->hour,t->minute);
        else
            putchar(c);
    }
}

int cmpclock(HM_TIME *t1, HM_TIME *t2)
{
    if (t1->hour > t2->hour)
        return 1;
    else if (t1->hour < t2->hour)
        return -1;
    else if (t1->minute > t2->minute )
        return 1;
    else if (t1->minute < t2->minute )
        return -1;
    else
        return 0;
}

void break_time(HM_TIME *t,int hm)
{
    t->hour = hm/100;
    t->minute = hm%100;
}

int bind_time(HM_TIME *t)
{
    return t->hour*100+t->minute;
}
```


図 7-19　スケジュール管理プログラムの一部「hm_time.c」

```
/*
        スケジュール管理プログラム用ライブラリ
*/
#include <stdio.h> ……………… printf()関数等用のヘッダファイルをインクルードする
#include <stdlib.h> ……………… malloc()関数等用のヘッダファイルをインクルードする
#include "hm_time.h" ……………… 時間処理プログラム用のヘッダファイルをインクルードする
#include "sch.h" ……………… スケジュール管理プログラム用のヘッダファイルをインクルードする

SCHEDULE *schtop; ……………… スケジュールデータの先頭項目を指すポインタ

void inssch( SCHEDULE *presch , HM_TIME *t , char *str )  リスト構造の中に一項目挿入する関数
{
    SCHEDULE *p;

    p = (SCHEDULE *) malloc( sizeof(SCHEDULE) ); ……………… 一項目分のメモリを割り当てる
    if (p == NULL) {
        printf("メモリが足りません\n");  } ……………… メモリが確保できなかった場合の処理
        exit(1);                         }
    }
    p->apotime.hour = t->hour;        } ……………… 一項目分のデータを代入する
    p->apotime.minute = t->minute;    }
    strcpy(p->apostr, str);
    p->next = presch->next;  } ……………… ポインタをつなぎかえてリスト構造に挿入する
    presch->next = p;        }
}

void delsch(SCHEDULE *presch, SCHEDULE *sch)  リスト構造から一項目削除する関数
{
    prtime_st(sch->apostr, &sch->apotime);  } ……………… 削除する項目を表示する
    printf("を削除します。\n");              }

    presch->next = sch->next; ……………… ポインタをつなぎかえてリスト構造から削除する
    free(sch); ……………… 一項目分のメモリを開放する
}

void searchsch( HM_TIME *t , char *str )  リスト構造を検索する関数
{
                ……………… リスト構造をたどるためのループ変数
    SCHEDULE *p,*presch; ……………… 1つ前の項目を指すポインタ
    int r;

    presch=(SCHEDULE *)&schtop; ……………… 1つ前の項目として schtop を指すようにする
    for ( p=schtop ; p ; p=p->next ) { ……………… リスト構造を順番にたどりながら繰り返す
        r = cmpclock( &p->apotime, t); ……………… 引数の時刻と項目に時刻を比較する

        if ( r == 0 ) {
            delsch( presch , p );  } ……………… 時刻が一致すれば削除する
            return;                }
        } else if ( r > 0 ) {
            inssch( presch , t , str );  } ……………… 引数の時刻より後の項目が見つかったら、
            return;                      }             そこに挿入する
        }
        presch = p; ……………… presch が今の項目を指すようにして、次の項目に進む
    }
    inssch( presch , t , str ); ……………… 全部の項目をたどっても引数の時刻よりあとの
}                                          項目が見つからなかったら末尾に追加する
```

図 7-20　スケジュール管理プログラムの一部「schsub.c」

```
/*
        スケジュール管理プログラム第1版
*/
#include <stdio.h> ・・・・・・・・・・printf()関数等用のヘッダファイルをインクルードする
#include "hm_time.h"・・・・・・・・・・時間処理プログラム用のヘッダファイルをインクルードする
#include "sch.h"・・・・・・・・・・スケジュール管理プログラム用のヘッダファイルをインクルードする

void prsch() スケジュールを表示する関数
{
    SCHEDULE *p;

    for ( p=schtop ; p ; p=p->next ) {
        prtime_st(p->apostr,&p->apotime);  }・・・・・・リスト構造をたどりながらすべての項目を表示する
        putchar('\n');
    }
    putchar('\n');・・・・・・・・・・最後にもう一度、改行する
}

#define MAXBUF 128 スケジュールを入力する関数
void readsch()
{
    int hm;
    char buf[MAXBUF]; ・・・・・・・・・・キーボードから入力する文字列を入れる配列
    HM_TIME t;

    for (;;) {
        gets(buf);・・・・・・・・・・キーボードから1行分読み込む
        if (buf[0]==0)  }・・・・・・・・・・入力がなければ終了する
            break;
        hm = atoi(buf); ・・・・・・・・・・文字列から数値へ変換する
        break_time(&t,hm);・・・・・・・・・・構造体に変換する
        gets(buf); ・・・・・・・・・・キーボードから1行分読み込む
        searchsch(&t,buf);・・・・・・・・・・リスト構造を検索する
        prsch();・・・・・・・・・・リスト構造を表示する
    }
}

main()
{
    HM_TIME t; ・・・・・・・・・・時刻を入れる構造体

    schtop = NULL; ・・・・・・・・・・リスト構造の先頭を何も指さない状態にセットする

    t.hour = 12;
    t.minute = 00;                                  }・・・・・・12:00の項目をあらかじめ入力しておく
    searchsch(&t,"お昼休みは ? からです。");

    prsch(); ・・・・・・・・・・スケジュールの表示
    readsch(); ・・・・・・・・・・スケジュールの入力
    prsch(); ・・・・・・・・・・スケジュールの表示
}
```

図7-21 スケジュール管理プログラムのメインプログラム「sch0.c」

1つは、時間を処理するモジュール「hm_time.c」です。モジュール内で使用する構造体のデータ型宣言、関数のプロトタイプを集めた「hm_time.h」をインクルードすることによって、他のモジュールから利用します。

2つめは、スケジュールを処理するモジュール「schsub.c」です。このモジュールで使用するデータ型や関数プロトタイプ宣言を集めたものが「sch.h」です。そして最後が、スケジュール情報を画面に表示したり、キーボードから入力したりする関数やmain()関数を定義した「sch0.c」です。「sch0.c」は、後で改良したプログラムsch.cを作成して正式版とするため、このようなファイル名にしました。

リスト構造を処理するアルゴリズム

スケジュール情報の各項目は、250ページの図7-16に示したschedule構造体型の変数に格納します。

グローバル変数schtopは、先頭の項目を指し示すポインタです。各項目を格納する構造体のメンバnextは、次の項目を指し示すポインタ型です。図7-23に、このプログラムの実行例と、その状態での変数の様子を示します。図のように、schtopからポインタをたどっていくことにより、すべての項目をながめることができます。

schtopはヘッダファイルsch.hでextern宣言されているので他のモジュールからもアクセスすることができます。

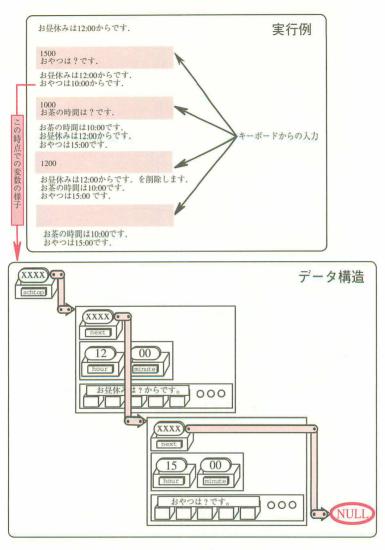



図 7-23　スケジュール管理プログラムの実行例とスケジュール表のデータ構造

NULL ポインタ

図 7-24 のように main() 関数の先頭で、schtop 変数に NULL という値を代入しています。NULL（ナル）は stdio.h で定義されているマクロ名で、「0」

に置き換えられます。

ポインタ変数には変数のアドレスを代入することはあっても、数値を代入することは、特別な場合を除いてありません。0はその特別な場合で、ポインタがどの変数も指し示していないことを示すために用います。

例題プログラムでは、図7-24のように最後の項目において、次の項目がないことを示すためにNULLを使っています。プログラムの実行開始時点では、1つも項目がありませんから、先頭を指すschtopにNULLを代入しておくのです。

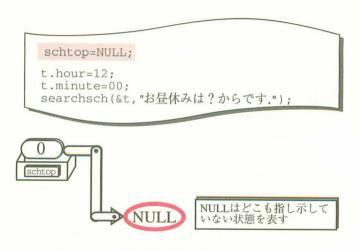



図7-24　NULLポインタ

リストの検索

searchsch()関数は、スケジュール表の一項目を受け取り、すでに登録されている項目の中から時刻が一致するものを検索する関数です。

一致する情報がなければ、新たな情報としてリスト構造の中に挿入します。一致する情報がある場合は、その情報を削除します。1つの関数で、情報の追加と削除を兼ねているわけです。

リスト構造を検索するには、図7-25のようにfor文を使って実現することができます。ポインタpに先頭要素を指すポインタを代入し、次の項目へ進むときはメンバnextの値を代入します。

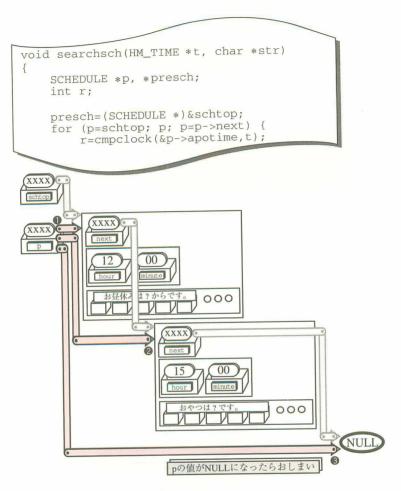



図 7-25　リスト構造の検索

ここで、終了判定式として「p」のみが使われていることに注目してください。130 ページで解説したように、C 言語では 0 以外の値はすべて真として処理されます。したがって、ポインタ p の値が 0 になったら繰り返しを終了します。ポインタ 0（NULL）は次の項目がないことを表すことにしていますから、最後の項目まで処理した後、繰り返しは終了します。

cmpclock()関数は、時刻の前後関係を調べる関数です。引数として渡した 2 つの時刻を比べて、その前後関係を図 7-26 のような値として返します。

searchsch()関数では、時刻の一致する項目が見つかれば、delsch()関数でその項目を削除します。そして、その時刻よりも後の時刻が見つかったら、その項目の前に新しい項目をinssch()関数で挿入します。どちらの条件にもあてはまらず繰り返しを終了した場合は、末尾に新しい項目を追加します。

cmpclock(時刻構造体1, 時刻構造体2)	時刻1＞時刻2なら1を返す 時刻1==時刻2なら0を返す 時刻1＜時刻2なら-1を返す

```
int cmpclock(HM_TIME *t1, HM_TIME *2)
```

図7-26　cmpclock()関数

リスト構造への挿入

inssch()関数は、リスト構造への挿入を行う関数です。図7-27のように、ポインタをつなぎ換えて新しい項目をリストに挿入します。

新しい項目の情報を格納する変数は、malloc()関数によって割り当てますが、これについては後で解説します。

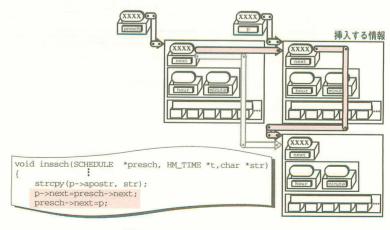



図7-27　リスト構造への挿入

リスト構造からの削除

delsch()関数は、リスト構造から項目を削除します。図7-28のように、ポインタをつなぎ換えて項目を削除します。

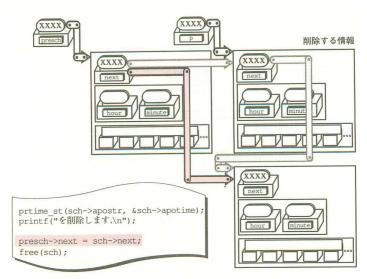



図7-28　リスト構造からの削除

だましのテクニック

図7-29は、例題プログラムの一部です。この(SCHEDULE *)というキャスト演算子による型変換には、いったいどういう意味があるのでしょうか。

```
void searchsch(HM_TIME *t, char *str)
{
    SCHEDULE *p, *presch;
    int r;

    presch = (SCHEDULE *)&schtop;
    for (p=schtop; p; p=p->next) {
        r = cmpclock(&(p->apotime), t);

        if (r ==0) {
            delsch(presch, p);
```

図7-29　型変換

ひとことでいえば、コンパイラをだましているのです。図7-30のように、変数schtopをあたかも構造体の一部であるかのように扱うことによって、処理を単純化しているのです。こうすることによって、ポインタpreschが1つ前の項目を指していても、先頭の項目を指すschtopを指していても同じように処理することができます。

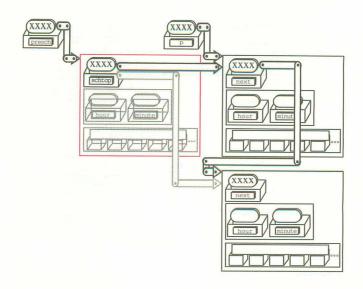



図7-30　だましのテクニック

構造体の1つめのメンバが、次の項目を指すメンバnextであるため、このような処理が可能となります。このような処理は、リスト構造を処理する際の典型的な「だましのテクニック」です。

これは、変数とメモリの関係を巧妙に利用したC言語ならではのテクニックです。他の言語では、このようなだましのテクニックは通用しません。ある意味では非常に便利な機能ですが、わかりにくい処理であることも確かです。

このようなテクニックを身につけてこそ、C言語をマスターしたといえるでしょう。しかし、プログラムのわかりやすさを考えれば、あまり濫用しないことも重要です。

malloc()関数と free()関数

図 7-27 や図 7-28 に示したような処理を実現するには、情報を格納する変数をあらかじめ用意しておくのではなく、プログラム実行中に新しく誕生させたり、消滅させたりしなければなりません。こうした処理のために用意されているのが、malloc()関数と free()関数です。

malloc()関数（エムアロックと読む）は、図 7-31 のようにヒープ領域から指定したバイト数のメモリを確保し、そのアドレスをポインタとして返します。malloc()関数の戻り値をポインタに代入すると、確保したメモリを変数として利用することができます。malloc()関数は、指定したサイズのメモリを確保できなかったときは NULL を返します。

free()関数は、ポインタとして渡されたメモリを変数として割り当てられていない状態に戻します。この処理をメモリを解放するといいます。

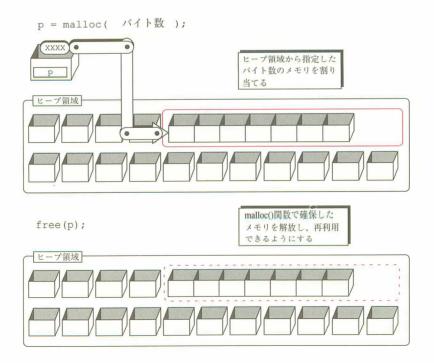



図 7-31 malloc()関数と free()関数

malloc()関数で確保した変数は、解放するまで他の変数に割り当てられることはありません。free()関数で解放すると、他の変数に割り当てられるようになります。

malloc()関数とfree()関数を利用することによって、メモリを有効活用することができます。例題プログラムのように、変数が必要になった時点でメモリを確保し、不要になった時点で解放することによって、その時点で必要な分だけしかメモリを消費しないようにするのです。

157ページで解説したように、ローカル変数は関数の実行開始時に自動的に確保され、実行終了時に解放されます。これに対し、malloc()関数を使って確保したメモリは、関数を終了しても解放されず、free()関数の呼び出しによってのみ解放されます。

sizeof 演算子

［書式］sizeof(変数名［または型名］)

malloc()関数を呼び出す際には、確保する変数に必要なメモリサイズを引数として渡さなければなりません。このために**sizeof 演算子**（サイズオブ）を使っています。sizeof 演算子は( )で囲んだデータ型を指定することにより、そのデータ型の変数のメモリサイズを求める演算子です。

また、malloc()関数の戻り値をポインタに代入するには、図7-33のようにキャストによって型変換をしなければなりません。malloc()関数の返す型はstdlib.hで宣言されていますが、一般には異なる型であるため、強制的に型変換して自分の使いたい型の変数へのポインタとして代入する必要があります*5。

```
void inssch( SCHEDULE *presch, HM_TIME *t, char *str)
{
    SCHEDULE *p;

    p = (SCHEDULE *) malloc( sizeof(SCHEDULE));
    if (p == NULL) {
        printf("メモ
        exit(1);
    }
    p->apotime.hour
    p->apotime.minute = t->minute;
    str
    p->
    pre
}
```

sizeof演算子は指定した型の変数に必要なメモリのバイト数を返す

next

hour minute

図 7-32　sizeof 演算子

```
void inssch( SCHEDULE *presch, HM_TIME *t, char *str)
{
    SCHEDULE *p;

    p = (SCHEDULE *) malloc( sizeof(SCHEDULE));
    if (p == NULL) {
        prin                                        );
        exit
    }
    p->apoti
    p->apotime.minute = t->minute;
    strcpy(p->apostr, str);
    p->next = presch->next;
    presch->next = p;
}
```

SCHEDULE型へのポインタ p に代入するためにはキャスト演算による型変換が必要

図 7-33　malloc()関数の戻り値のキャスト

＊5 stdlib.h で void ＊型として宣言されていれば、この型変換は必要ありません。また、ANSI に準拠していない処理系や OS では、stdlib.h ではなく malloc.h で宣言されているものもあります。

変数名の付け方2

40ページのコラムで解説した規則さえ守れば、変数には自由に名前を付けることができます。しかし、例題プログラムをいくつか読むうちに、変数名の付け方にはちょっとしたルールがあることに気づかれたでしょうか。

実は、変数名の付け方には、図に示したように、慣習的に用いられているルールがあります。あまりこだわる必要はありませんが、知っているとベテランプログラマーの書いたプログラムが読みやすくなります。

変数名	小文字を使う
i　j　k	0，1，2と増えるループ変数
c　ch	char 型変数
p　ptr	ポインタ
str	文字列
	ループ変数や短い間しか使わない変数には1～3文字程度の短い名前を付け，比較的長い範囲で使う変数には，もっと長い単語を使う．

マクロ名	大文字を使う
TABLESIZE, NULL, FILE	

変数名のルール

7.3

プログラムの実行環境

プログラムの実行環境

プログラムの実行環境とは、実行するプログラムにとってのまわりの環境のことです。もちろん、環境といっても日当たりが良いとか公園が近いとかいうことではありません。コンピュータの中で動作するプログラムにとっての環境ですから、コンピュータの内部状態や周辺機器との入出力のことを意味します。たとえば図7-34のような条件をプログラムの実行環境と呼びます。

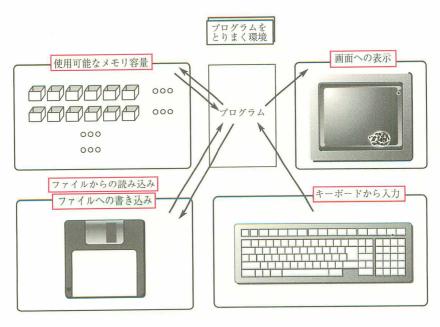



図7-34　プログラムの実行環境

プログラミング技術を向上させるには、前節で解説したアルゴリズムとデータ構造を習得するとともに、こうした環境の利用方法を習得しなければなりません。

ライブラリ関数の中には、実行環境を利用するための関数がたくさん用意されています。図7-35にこうしたライブラリ関数の一部を示しました。本書ではこれらの関数の使い方はあまり解説していませんが、ぜひマニュアルやプログラム集を通じて勉強してください。

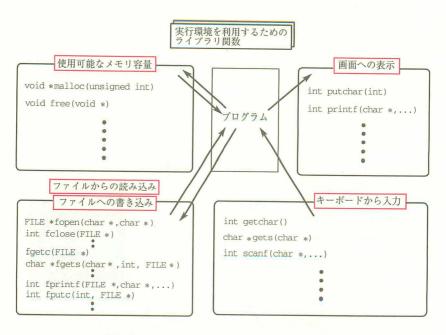



図7-35　環境を利用するためのライブラリ関数

オペレーティングシステム

実行環境を利用するためには、オペレーティングシステムの働きを知ることが非常に役に立ちます。オペレーティングシステムの機能を勉強することで、実行環境の利用方法は明らかになってくるでしょう。

みなさんの使っているコンピュータでは、MS-DOSやUNIXなどのOSが動作しています。OSは「オペレーティングシステム(Operating System)」

の略で、プログラムの実行環境を管理し、周辺装置との入出力を中継してくれるソフトウェアの集合体のことです。

図 7-36 のように、オペレーティングシステムはメモリ内に常駐し、コンピュータ全体を制御し、管理しています。ライブラリ関数の多くは、オペレーティングシステムを呼び出すことによって、周辺装置の制御などを行っています。

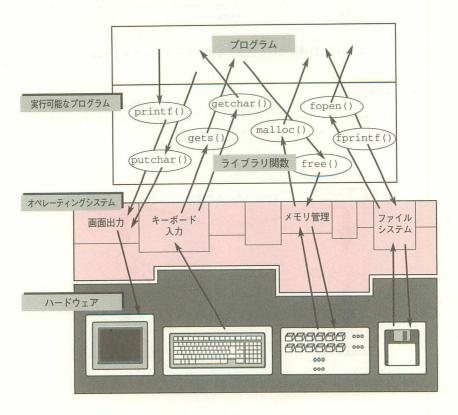



図 7-36　オペレーティングシステム

オペレーティングシステムの役割は、実行環境の利用方法を提供することです。コンピュータの機種や周辺装置の種類が異なれば、機械的なレベルでの操作方法も異なります。しかし、オペレーティングシステムの呼び出し方法は統一されており、こうした違いを意識する必要はありません。

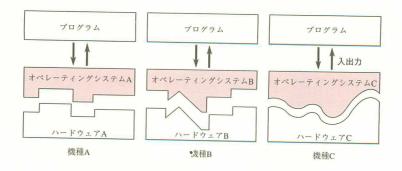



図7-37　機種の違いを吸収するオペレーティングシステム

ファイル入出力

ファイル入出力を例に、実行環境の利用方法を解説しましょう。ファイルへの入出力は、図7-38のような3段階の手順で行います。

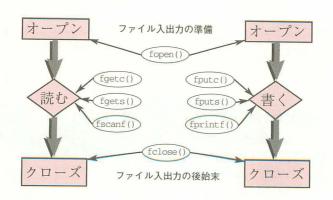



図7-38　ファイル入出力の手順

ファイルのオープンは、ファイル入出力のための準備作業です。図7-39に、ファイルオープンを行うfopen()関数の呼び出し方法と、その役割を示します。

ファイルオープンでは、指定されたファイル名のファイルが存在するかどうかを確認し、ディスク上の格納位置などの管理情報をメモリ内に作成します。書き込みモードの場合、ファイルが存在しなければ作成します。

```
FILE *fopen(ファイル名, アクセスモード);
```

戻り値	FILE構造体へのポインタ ファイルが存在しない，または作成できない場合は NULLを返す
アクセスモード	"r" ―― 読み込みのためにオープン "w" ―― 書き込みのためにオープン "a" ―― ファイル末尾に追加書き込みするためにオープン

```
main()
{
    char *fnam;
    FILE *fp;
    HM_TIME t;

    schtop = NULL;
    fnam = "sch.dat";

    fp = fopen(fnam, "r");
    it (fp == NULL) {
```

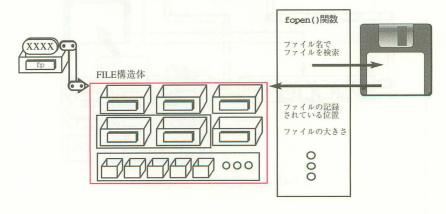



図7-39　ファイルオープン

fopen()関数は、ファイル管理情報構造体へのポインタを戻り値として返します。この構造体は、FILE 型として stdio.h の中で定義されています。FILE

型構造体へのポインタ、つまりFILE *型のポインタのことを**ファイルポインタ**と呼びます。

ファイルオープンに成功すると、ファイルから情報を読みだしたり書き込んだりすることができるようになります。読み込みや書き込み手順では、図7-40のように、ファイルポインタを使ってファイルを指定します。ファイル名が必要なのは、オープン時だけです。

```
if ( fgets(buf, MAXBUF, fp) == NULL || buf[0] =='\n')
    break;
hm = atoi(buf);
break_time(&t, hm);
fgets(buf, MAXBUF,fp);
searchsch(&t, buf);
```

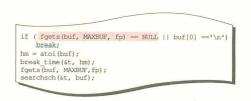
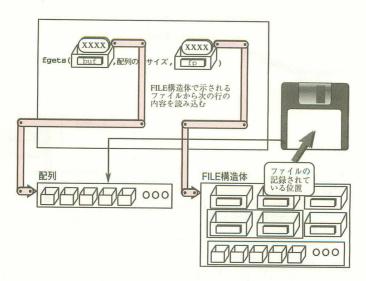



図7-40　fprintf()関数、fgets()関数

入出力が終了すると、ファイルをクローズしなければなりません。クローズは、ファイルの管理情報をディスクに書き込み、それを格納するために確保したメモリを解放するといった、後始末の作業です。クローズを行わなければファイルは正しく作成されず、書き込んだ情報は失われてしまいます。

```
} else {
    readsch(fp);
    fclose(fp);
}
```

図 7-41　fclose()関数

ファイル入出力の例題プログラム（スケジュール管理プログラム）

次に示した図 7-42 は、前節で作成したスケジュール管理プログラムにファイル入出力機能を追加したプログラムです。ソースファイル sch0.c の部分を sch.c に置き換えてしまいます。

このプログラムでファイル入出力を行っている部分を、わかりやすく囲みで示しました。ファイルからスケジュールデータを読み出す部分、ファイルに書き込む部分で 3 段階の手順に従って処理していることがわかります。

```
#include <stdio.h>
#include <stdlib.h>
#include "hm_time.h"
#include "sch.h"

void prsch()            スケジュールを表示する関数
{
    SCHEDULE *p;
    int n;

    for ( p=schtop ; p ; p=p->next )
        prtime_st(p->apostr,&p->apotime); ……… fgets()関数では行末の改行もデータとして
    putchar('\n');                             返されるので、ここで改行する必要はない
}

#define MAXBUF 128
void readsch( FILE *fp )        スケジュールをファイルから読み込む関数
{
        int hm;
        char buf[MAXBUF];
        HM_TIME t;
                                      ……ファイルから1行分データを入力する
        for (;;) {
                if ( fgets(buf,MAXBUF,fp)== NULL || buf[0]=='\n')
                    break;
                hm = atoi(buf);      読み込み 2
                break_time(&t,hm);
                fgets(buf,MAXBUF,fp);
                searchsch(&t,buf);
        }
}

void writesch( FILE *fp )        スケジュールをファイルに書き出す関数
{
    SCHEDULE *p;
                                          書き込み 2
    for ( p=schtop ; p ; p=p->next ) {
        fprintf( fp , "%d\n", bind_time( &p->apotime ) ); ……… ファイルにデータを書き出す
        fprintf( fp , "%s", p->apostr );
```

```
    }
}

main()
{
    char *fnam;
    FILE *fp;
    HM_TIME t;

    schtop = NULL;
    fnam = "sch.dat";  読み込み 1

    fp = fopen( fnam , "r" );  ……ファイルを読み込みモードでオープンする
    if ( fp==NULL ) {  ……ファイルポインタが NULL ならファイルが存在しない
        printf("新規スケジュールです。\n", fnam );

        t.hour = 12;
        t.minute = 00;
        searchsch(&t,"お昼休みは ? からです。\n");

    } else {
        readsch(fp);  ……ファイルからデータを読み込む
        fclose( fp );  ……ファイルをクローズする

    }        読み込み 3

    prsch();
    readsch( stdin );  ……標準入力（キーボード）からスケジュールを入力する
    prsch();        書き込み 1

    fp = fopen( fnam , "w" );  ……ファイルを書き込みモードでオープンする
    if ( fp==NULL ) {
        printf("ファイル%sをオープンできません。\n", fnam );  } ……ディスクが一杯か、リードオンリーの場合
        exit(1);
    }
    writesch( fp );  ……スケジュールをファイルに書き出す
    fclose( fp );  ……ファイルをクローズする
}        書き込み 3
```

図 7-42　sch.c

このプログラムの実行例を、図 7-43 に示します。合わせて、入力したスケジュール情報がファイルに書き込まれている様子を示します（図 7-44）。

```
新規スケジュールです。
お昼休みは 12:00 からです。

1500↵
おやつは ? です。↵
1000↵
お茶の時間は ? です。↵
↵
お茶の時間は 10:00 です。
お昼休みは 12:00 からです。
おやつは 1500 です。
```

図 7-43　スケジュール管理プログラムの実行例

ファイルsch.datに書き込まれた内容

```
1000
お茶の時間は ? です。
1200
お昼休みは ? からです。
1500
おやつは ? です。
```

図 7-44 スケジュールデータファイルに書き込まれた情報

コマンドラインパラメータ

次に、コマンドラインパラメータを例に、実行環境の利用方法を解説しましょう。

コマンドを実行するときには、ファイル名やオプションなど、**コマンドラインパラメータ**を指定します。コマンドラインパラメータは、プログラムが実行される前にユーザーがプログラムに対して与える情報です。OSはこれを受け取り、プログラムに引き渡してくれます。

コマンドラインパラメータは、ライブラリ関数を通して受け取るのではなく、main()関数の引数として渡されます。main()関数にはこれまで引数がないかのように扱ってきましたが、実は、図7-45に示すように2つの引数を持っています。

main()関数の引数のうち、1つめはパラメータの数「argc」です。そしてもうひとつは、パラメータ文字列を渡すための引数「argv」で、argvの型は文字列へのポインタの配列です。

ポインタの配列という型は一見難しそうに思えますが、図7-45のようにポインタを並べただけもので、複数の文字列をまとめて扱う際にはよく使うデータ構造です。パラメータは図のように1つ1つ文字列に分解されて渡されます。

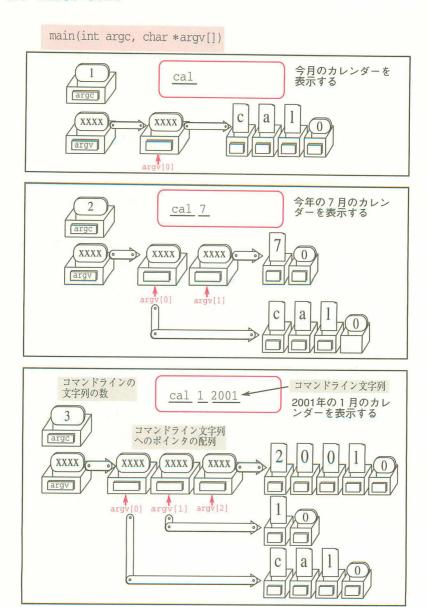



図7-45　パラメータ文字列

本項の例題プログラムは、「カレンダー表示プログラム」です。実行可能プログラムを cal コマンドとして実行することにすると、図7-45のようなコマ

ンドラインパラメータで年や月を指定します。

カレンダー表示プログラムを、図7-46に示します。日付計算を行う関数は、134ページですでに作成していまから、234ページの図7-3のようにcalcdate.cを作成してリンクしてください。ヘッダファイルcalcdate.hはcalcdate.cで定義した関数のプロトタイプ宣言を集めたファイルです。このファイルは図7-10に示してあります。

カレンダーを表示する処理自体はとても簡単です。難しいのは、その月のはじまる曜日を求めることです。曜日を求めるにはいろいろな方法があるでしょうが、ここではldays()関数によって求めた西暦1年1月1日からの日数を使うことにします。曜日はかならず7日で1周しますから、日数を7で割った余りから曜日を知ることができます。このプログラムの実行例を図7-47に示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> ·········· time()関数、localtime()関数用のヘッダファイル
#include "calcdate.h"

int dayofweek(int y,int m,int d) ·········· 曜日を求める関数
{
    int w;

    w = ldays(y,m,d) % 7; ·········· 0…日曜、1…月曜、……、6…土曜
    return w;
}

void dispcal(int y,int m) ·········· カレンダーを表示する関数
{
    int w,i,maxd;

    printf("%4d年 %2d月 \n",y,m);
    printf(" 日  月  火  水  木  金  土\n");
    w=dayofweek(y,m,1); ·········· 1日の曜日を求める
    for (i=0;i<w;i++) ·········· 1日の曜日にくるまで空日を表示
        printf("    ");
    maxd=mdays(y,m); ·········· その月の日数を求める
    for (i=1;i<=maxd;i++) { ·········· 月の日数くり返す
        printf(" %2d ",i); ·········· 日付を表示する
        w = (w+1) % 7; ·········· 次の曜日を求める
        if (w==0) ·········· 次が日曜なら改行する
            printf("\n");
    }
```

```
        printf("\n\n");
    }

    main( int argc, char *argv[] )
    {
        int year, month;
        time_t t;           ……現在の時刻を得るための変数
        struct tm *tp;

        time(&t);              ……現在の時刻を得る
        tp = localtime(&t);      詳細はライブラリ関数のマニュアルを見てください

        if ( argc==1 ) {……コマンドライン引数が1個の場合［cal　］
            year = 1900 + tp->tm_year;   ……今月をセット
            month = tp->tm_mon +1;
        } else if ( argc==2 ) {……コマンドライン引数が2個の場合［cal　月］
            year = 1900 + tp->tm_year;……今年をセット
            month = atoi(argv[1]);……引数の月をセット
        } else if ( argc==3 ) {……コマンドライン引数が3個の場合
            year = atoi(argv[2]);……第2引数を年にセット
            month = atoi(argv[1]);……第1引数を月にセット
        } else {……それ以外の場合
            printf("usage: calendar month [year]\n");
            exit(1);               ……エラーメッセージを表示して終了
        }

        dispcal(year,month);……カレンダーを表示する

    }
```

図7-46　カレンダー表示プログラム cal.c

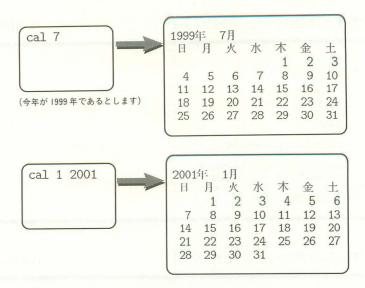



図7-47 カレンダー表示プログラムの実行例

〈本章で取り上げたプログラム〉プログラム7-1 sch.h

```
/*
    スケジュール管理プログラム用ヘッダファイル
*/


#define MAXAPOSTR 128
struct schedule {
    struct schedule *next;
    HM_TIME     apotime;
    char apostr[MAXAPOSTR];
};
typedef struct schedule SCHEDULE;

void delsch(SCHEDULE *presch, SCHEDULE *sch);
void inssch( SCHEDULE *presch , HM_TIME *t , char *str );
void searchsch( HM_TIME *t , char *str );

extern SCHEDULE *schtop;
```

注記：
- struct schedule { … }; の部分 → スケジュールの一項目のデータ型の宣言
- typedef struct schedule SCHEDULE; → 型名の置き換え
- delsch / inssch / searchsch の宣言 → スケジュール管理関数のプロトタイプ宣言
- extern SCHEDULE *schtop; → グローバル変数の宣言

〈本章で取り上げたプログラム〉プログラム7-2　hm_time.h

```
/*
   時間構造体定義ヘッダ
*/

struct hm_time {  ……時間構造体のデータ型宣言
    int hour;
    int minute;
};
typedef struct hm_time HM_TIME;  ……型名の置き換え

void prtime_st(char *str, HM_TIME *t);   ┐
int cmpclock(HM_TIME *t1, HM_TIME *t2);  │……時間処理関数のプロトタイプ宣言
void break_time(HM_TIME *t,int hm);      │
int bind_time(HM_TIME *t);               ┘
```

〈本章で取り上げたプログラム〉プログラム7-3　hm_time.c

```
#include <stdio.h>  ……printf()関数等用のヘッダファイルをインクルードする
#include "hm_time.h"  ……時間処理プログラム用のヘッダファイルをインクルードする

void prtime_st(char *str, HM_TIME *t)  ……文字列中の'?'を時刻に置き換えて表示する関数
{                                        6章で作成したもの
    char c;

    while(*str) {
        c = *str++;
        if (c=='?')
            printf("%2d:%02d",t->hour,t->minute);
        else
            putchar(c);
    }
}

int cmpclock(HM_TIME *t1, HM_TIME *t2)  ……2つの時刻を比較する関数
{
    if (t1->hour > t2->hour)  ……時刻1 > 時刻2 なら1を返す
        return 1;
    else if (t1->hour < t2->hour)  ……時刻1 < 時刻2 なら-1を返す
        return -1;
    else if (t1->minute > t2->minute )  ……時刻1 > 時刻2 なら1を返す
        return 1;
    else if (t1->minute < t2->minute )  ……時刻1 < 時刻2 なら-1を返す
        return -1;
    else  ……時刻1 == 時刻2 なら0を返す
        return 0;
}

void break_time(HM_TIME *t,int hm)  ……100倍法で表された時刻を構造体に代入する関数
{
    t->hour = hm/100;
    t->minute = hm%100;
}
```

```
int bind_time(HM_TIME *t) ………………………構造体から100倍法で表された時刻に変換する関数
{
    return t->hour*100+t->minute;
}
```

〈本章で取り上げたプログラム〉プログラム7-4　schsub.c

```
/*
        スケジュール管理プログラム用ライブラリ
*/
#include <stdio.h> ……………………………… printf()関数等用のヘッダファイルをインクルードする
#include <stdlib.h> ……………………………… malloc()関数等用のヘッダファイルをインクルードする
#include <string.h> ……………………………… strcpy()関数等用のヘッダファイルをインクルードする
#include "hm_time.h" ……………………………時間処理プログラム用のヘッダファイルをインクルードする
#include "sch.h" ……………………………………スケジュール管理プログラム用のヘッダファイルをインクルードする

SCHEDULE *schtop; ………………………………………スケジュールデータの先頭項目を指すポインタ

void inssch( SCHEDULE *presch , HM_TIME *t , char *str )リスト構造の中に一項目挿入する関数
{
    SCHEDULE *p;

    p = (SCHEDULE *) malloc( sizeof(SCHEDULE) ); …………………一項目分のメモリを割り当てる
    if (p == NULL) {
        printf("メモリが足りません\n");    …………メモリが確保できなかった場合の処理
        exit(1);
    }
    p->apotime.hour = t->hour;
    p->apotime.minute = t->minute;   …………………一項目分のデータを代入する
    strcpy(p->apostr, str);
    p->next = presch->next;   ………………………ポインタをつなぎかえてリスト構造に挿入する
    presch->next = p;
}

void delsch(SCHEDULE *presch, SCHEDULE *sch) ………リスト構造から一項目削除する関数
{
    prtime_st(sch->apostr, &sch->apotime);   …………削除する項目を表示する
    printf("を削除します。\n");

    presch->next = sch->next; ……………………………ポインタをつなぎかえてリスト構造から削除する
    free(sch); ……………………………………………………一項目分のメモリを開放する
}

void searchsch( HM_TIME *t , char *str ) ………………リスト構造を検索する関数
{
    SCHEDULE *p,*presch; ……………………………………1つ前の項目を指すポインタ
    int r;      …………………………………………………リスト構造をたどるためのループ変数

    presch=(SCHEDULE *)&schtop; ……………………………1つ前の項目として schtop を指すようにする
    for ( p=schtop ; p ; p=p->next ) { ………………リスト構造を順番にたどりながら繰り返す
        r = cmpclock( &p->apotime , t); ………………引数の時刻と項目の時刻を比較する

        if ( r == 0 ) {
            delsch( presch , p ); ………………………時刻が一致すれば削除する
```

```
            return;
        } else if ( r > 0 ) {
            inssch( presch , t , str ); ……………………引数の時刻より後の項目が見かったら、そこに挿入する
            return;
        }
        presch = p; ……………………………………………… presch が今の項目を指すようにして、次の項目に進む
    }
    inssch( presch , t , str ); ………………………………全部の項目をたどっても引数の時刻よりあとの項目が見
}                                                    つからなかったら末尾に追加する
```

〈本章で取り上げたプログラム〉プログラム 7-5　sch0.c

```
/*
        スケジュール管理プログラム第1版
*/
#include <stdio.h> ……………………………………………… printf()関数等用のヘッダファイルをインクルードする
#include "hm_time.h"          ……………………………………時間処理プログラム用のヘッダファイルをインクルードする
#include "sch.h"          ……………………………………………スケジュール管理プログラム用のヘッダファイルをインクルードする

void prsch() ……………………………………………………………スケジュールを表示する関数
{
    SCHEDULE *p;

    for ( p=schtop ; p ; p=p->next ) { ……………………リスト構造をたどりながらすべての項目を表示する
        prtime_st(p->apostr,&p->apotime);
        putchar('\n');
    }
    putchar('\n'); …………………………………………………最後にもう一度、改行する
}

#define MAXBUF 128
void readsch() ……………………………………………………スケジュールを入力する関数
{
    int hm;
    char buf[MAXBUF]; ……………………………………………キーボードから入力する文字列を入れる配列
    HM_TIME t;

    for (;;) {
        gets(buf); …………………………………………………キーボードから１行分読み込む
        if (buf[0]==0) ……………………………………………入力がなければ終了する
            break;
        hm = atoi(buf); ……………………………………………文字列から数値へ変換する
        break_time(&t,hm); ………………………………………構造体に変換する
        gets(buf); …………………………………………………キーボードから１行分読み込む
        searchsch(&t,buf); ………………………………………リスト構造を検索する
        prsch(); ………………………………………………………リスト構造を表示する
    }
}

main()
{
    HM_TIME t; ……………………………………………………時刻を入れる構造体

    schtop = NULL; ………………………………………………リスト構造の先頭を何も指さない状態にセットする
```

```
    t.hour = 12;
    t.minute = 00;
    searchsch(&t,"お昼休みは ? からです。");      ……12:00の項目をあらかじめ入力しておく

    prsch();      ……スケジュールの表示
    readsch();    ……スケジュールの入力
    prsch();      ……スケジュールの表示
}
```

〈本章で取り上げたプログラム〉プログラム7-6　sch.c

```
/*
        スケジュール管理プログラム第2版
*/

#include <stdio.h>      ……printf()関数等用のヘッダファイル
#include <stdlib.h>     ……atoi()関数用のヘッダファイル
#include <string.h>     ……strcpy()関数用のヘッダファイル
#include "hm_time.h"
#include "sch.h"

void prsch()      ……スケジュールを表示する関数
{
    SCHEDULE *p;

    for ( p=schtop ; p ; p=p->next )
        prtime_st(p->apostr,&p->apotime);      ……fgets()関数では行末の改行もデータとして返されるので、ここで改行する必要はない
    putchar('\n');
}

#define MAXBUF 128
void readsch( FILE *fp )      ……スケジュールをファイルから読み込む関数
{
        int hm;
        char buf[MAXBUF];
        HM_TIME t;

        for (;;) {
                if ( fgets(buf,MAXBUF,fp)== NULL || buf[0]=='\n')
                    break;      ……ファイルから1行分データを入力する
                hm = atoi(buf);
                break_time(&t,hm);
                fgets(buf,MAXBUF,fp);      ……(同上)
                searchsch(&t,buf);
        }
}

void writesch( FILE *fp )      ……スケジュールをファイルに書き出す関数
{
    SCHEDULE *p;

    for ( p=schtop ; p ; p=p->next ) {
        fprintf( fp , "%d\n", bind_time( &p->apotime ) );      ……ファイルにデータを書き出す
        fprintf( fp , "%s", p->apostr );
    }
```

```
}

main()
{
    char *fnam;
    FILE *fp; ………… ファイルポインタを入れる変数
    HM_TIME t;

    schtop = NULL;
    fnam = "sch.dat"; ………… ポインタ型変数 fnam にファイル名の文字列"sch.dat"を指すポインタをセットしておく
    fp = fopen( fnam , "r" ); ………… ファイルを読み込みモードでオープンする
    if ( fp==NULL ) { ………… ファイルポインタが NULL ならファイルが存在しない
        printf("新規スケジュールです。\n", fnam );

        t.hour = 12;
        t.minute = 00;
        searchsch(&t,"お昼休みは ? からです。\n");

    } else {
        readsch(fp); ………… ファイルからスケジュールデータを読み込む
        fclose( fp ); ………… ファイルをクローズする
    }

    prsch();
    readsch( stdin ); ………… 標準入力(キーボード)からスケジュールデータを入力する
    prsch();

    fp = fopen( fnam , "w" ); ………… ファイルを書き込みモードでオープンする
    if ( fp==NULL ) { ………… ファイルポインタが NULL ならファイルをオープンできない
        printf("ファイル%sをオープンできません。\n", fnam ); } ………… ディスクが一杯か、
        exit(1);                                                    リードオンリーの場合
    }
    writesch( fp ); ………… スケジュールデータをファイルに書き込む
    fclose( fp ); ………… ファイルをクローズする
}
```

〈本章で取り上げたプログラム〉プログラム7-7　calcdate.h

```
int isleapyear(int y);                                 }
int mdays(int year, int month);                        } ………… 日付の計算を行う関数の
int ydays(int year, int month, int date);              }          プロトタイプ宣言
long int ldays(int y,int m,int d);                     }
```

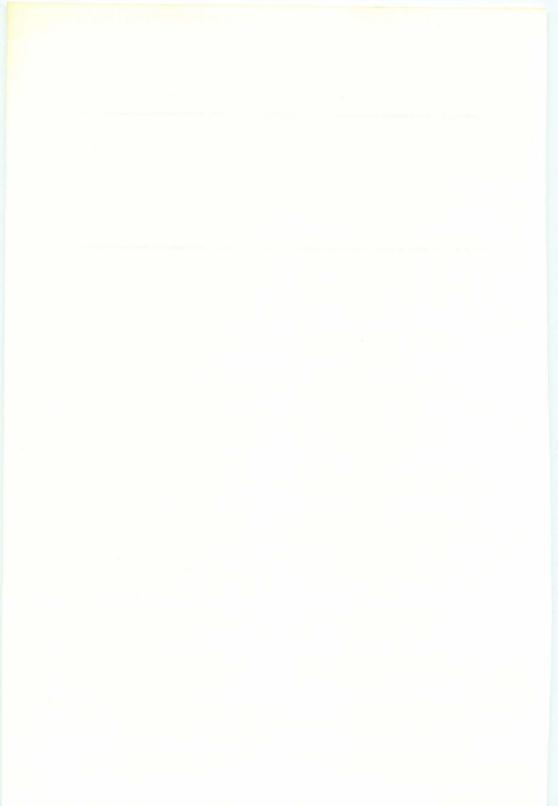
〈本章で取り上げたプログラム〉プログラム7-8　cal.c

```
#include <stdio.h>
#inlcude <stdlib.h>
#include <time.h> ………… time()関数、localtime()関数用のヘッダファイル
#include "calcdate.h"
```

```
int dayofweek(int y,int m,int d) ……曜日を求める関数
{
    int w;

    w = ldays(y,m,d) % 7; ……0…日曜、1…月曜、……、6…土曜
    return w;
}

void dispcal(int y,int m) ……カレンダーを表示する関数
{
    int w,i,maxd;

    printf("%4d年 %2d月\n",y,m);
    printf(" 日  月  火  水  木  金  土\n");
    w=dayofweek(y,m,1); ……1日の曜日を求める
    for (i=0;i<w;i++) ……1日の曜日にくるまで空日を表示
        printf("    ");
    maxd=mdays(y,m); ……その月の日数を求める
    for (i=1;i<=maxd;i++) { ……月の日数くり返す
        printf(" %2d ",i); ……日付を表示する
        w = (w+1) % 7; ……次の曜日を求める
        if (w==0) ……次が日曜なら改行する
            printf("\n");
    }
    printf("\n\n");
}

main( int argc, char *argv[] )
{
    int year, month;
    time_t t; ……現在の時刻を得るための変数
    struct tm *tp;

    time(&t);               } ……現在の時刻を得る
    tp = localtime(&t);     }    詳細はライブラリ関数のマニュアルを見てください

    if ( argc==1 ) { ……コマンドライン引数が1個の場合［cal ］
        year = 1900 + tp->tm_year;  } ……今月をセット
        month = tp->tm_mon +1;      }
    } else if ( argc==2 ) { ……コマンドライン引数が2個の場合［cal 月］
        year = 1900 + tp->tm_year; ……今年をセット
        month = atoi(argv[1]); ……引数の月をセット
    } else if ( argc==3 ) { ……コマンドライン引数が3個の場合［cal 月 年］
        year = atoi(argv[2]); ……第2引数を年にセット
        month = atoi(argv[1]); ……第1引数を月にセット
    } else { ……それ以外の場合
        printf("usage: calendar month [year]\n");
        exit(1); ……エラーメッセージを表示して終了
    }

    dispcal(year,month); ……カレンダーを表示する

}
```

〈本章で取り上げたプログラム〉プログラム7-9　calcdate.c

```
#include <stdio.h>
#include "calcdate.h"

int isleapyear(int y) ……指定した年が閏年なら1を返し、閏年でなければ0を返す関数
{
    if ( y%4==0 && y%100!=0 || y%400==0 ) ……4で割った余りが0ならば4で割り切れることを利用する
        return 1;
    else
        return 0;
}

int mdays(int year, int month) ……指定した月の日数を返す関数
{
    int days;

    switch(month) { ……月によって処理を分散させる
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:          } ……大の月の日数は31日
        days=31;
        break;
    case 4: case 6: case 9: case 11:   } ……小の月の日数は30日
        days=30;
        break;
    case 2:
        days=28+isleapyear(year);      } ……2月は閏年でなければ28日
        break;                               閏年ならば29日
    default: ……1から12まで以外の数値が与えられた時の処理
        printf("mdays(%d,%d):parameter error\n",year,month);
        break;
    }
    return days;
}

int ydays(int year, int month, int date) ……1月1日からの総日数を返す関数
{
        int days,i;

        days=0;
        for (i=1 ; i<month ; i++)
                days+=mdays(year,i); ……1月から前月までの日数を合計する
        days += date; ……今月の日数を加える
        return days;
}
```

```
#include "calcdate.h"

long int ldays(int y,int m,int d) ……西暦1年1月1日からの総日数を返す関数
{
    long int l; ……long int 型の変数 l を宣言する

    l=((long int)y-1)*365+(y-1)/4-(y-1)/100+(y-1)/400 + ydays(y,m,d);
                          ……365×昨年の西暦+昨年までの閏年の数+今年1月1日からの日数
    return l;
}
```

「Appendix」

1　主要処理系の紹介とコンパイルの実際
2　関数リファレンスの使い方
3　演算子書式一覧
4　演算子優先順位一覧
5　文法要覧
6　キーワード一覧
7　文字キャラクタセット
8　参考文献一覧

1 主要処理系の紹介とコンパイルの実際

● Microsoft C Version 6.0(Professional Development System)

発売元：マイクロソフト株式会社	
連絡先：〒160 東京都新宿区新宿 7-5-25 Kビルディング　TEL(03)3363-1201	
価　格：98,000円(税別 1991年4月1日現在)	
動作環境	CPU：80286、または80386
	OS：MS-DOS ver.3.1以上、またはOS/2 Ver.1.1A以上
	メモリ：640KB以上(MS-DOS)/1MB以上(OS/2)要ハードディスク（6MB以上）
	EMSメモリ：1MB以上（なくても可）
構成	マニュアル：3冊（Install Guide, Programming Guide, Reference）
	フロッピーディスク：8枚(2HD)
インストール方法	「Setup Disk」に入っている「setup.exe」を実行してください。詳しくはマニュアル(Install Guide)を参照してください。
コンパイル手順	MS-DOSのコマンドラインからコンパイルする方法を次に示します。

```
B:¥>type addclock.c↵ ......................... addclock.c の内容を表示する
#include <stdio.h>

main()
{
	int hour,minute;

	hour = 9 + 1;
	minute = 45 + 25;

	if (minute >= 60) {
		hour = hour+1;
		minute = minute-60;
	}

	printf("%d:%d¥n",hour,minute);
}

B:¥>
```

⬇

```
B:¥>cl addclock.c⏎ ………………………… cl addclock.c⏎と入力してコンパイラを起動する
Microsoft (R) C Optimizing Compiler Version 6.00A
Copyright (c) Microsoft Corp 1984-1990. All rights reserved.

addclock.c

Microsoft (R) Segmented-Executable Linker  Version 5.10
Copyright (C) Microsoft Corp 1984-1990.  All rights reserved.

Object Modules [.OBJ]: addclock.obj /farcall
Run File [addclock.exe]: "addclock.exe" /noi
List File [NUL.MAP]: NUL
Libraries [.LIB]:
Definitions File [NUL.DEF]: ;

B:¥>
```

…………コンパイラが自動的にリンカを起動し実行形式ファイル addclock.exe を作る

```
B:¥>addclock⏎ ………………………………… addclock⏎でプログラムを実行する
11:10

B:¥>
```

● Quick C Version 2.0

発売元：マイクロソフト株式会社	
連絡先：〒160 東京都新宿区新宿 7-5-25 Kビルディング　TEL(03)3363-1201	
価　格：20,000 円(税別 1991 年 4 月 1 日現在)	
動作環境	OS：MS-DOS Ver.2.0 以上
	メモリ：640KB 以上 ハードディスク、またはフロッピーディスクドライブ 2 台
構成	マニュアル：5 冊（Up and Running、Tool Kit、C for Yourself、Graphics Library Reference, Quick Reference）
	フロッピーディスク：4 枚(2HD)/6 枚(2DD)
インストール方法	「セットアップ/コンパイラディスク」に入っている「setup.exe」を実行してください。詳しくはマニュアル(Up and Running)を参照してください。
コンパイル手順	Quick C では、エディタなどを含めた統合環境が提供されています。統合環境では、エディタの画面からメニュー選択によってコンパイル作業を実行したり、デバック作業を行うことができます。以下にこの環境で、コンパイルする方法を示します。

```
F/ファイル E/編集 U/表示 S/サーチ M/メイク R/実行 D/修正 U/補助 O/環境 H/ヘルプ
                         B:¥ADDCLOCK.C
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1;
    minute = 45 + 25;

    if (minute >= 60) {
        hour = hour+1;
        minute = minute-60;
    }

    printf("%d:%d¥n",hour,minute);
}

<f1=ヘルプ> <GRPH=メニュー> <SHIFT+f5=再実行>                00016:002
```

メニューバーの【F/ファイル】から【O/読込】を選択し、ソースプログラムを呼び出す。

メニューバーの【M/メイク】から【C/コンパイル】を選択すると、コンパイル作業が実行され、同じく【B/プログラムファイル】を選択するとリンクされ addclock.exe ができる

```
F/ファイル  E/編集  V/表示  S/サーチ  M/メイク  R/実行  D/修正  U/補助  O/環境  H/ヘルプ
                                    B:¥ADDCLOCK.C
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1
    minute = 45

    if (minute >
        hour = h
        minute =
    }

    printf("%d:%
}
```

```
Compiling: addclock.c

            Lines   Errors   Warnings
Current:      16       0         0
Total:       181       0         0

      中断するときは，ESCキーを押します
```

コンパイル中の画面

```
<f1=ヘルプ> <GRPH=メニュー> <SHIFT+f5=再実行>                00016:002
```

```
B:¥>qc addclock.c
 ADDCLOCK.EXE
11:10

実行時間 = 00:00:00.00.  プログラム終了 (6).  何かキーを押してください
```

メニューバーの【R/実行】から【G/実行】を選択すると、addclock.exe が実行される。

● TURBO C Version 2.0

<table>
<tr><td colspan="2">発売元：株式会社ボーランドジャパン</td></tr>
<tr><td colspan="2">連絡先：株式会社マイクロソフトウェアアソシエイツ
〒107 東京都港区南青山 7-8-1 小田急南青山ビル　TEL(03)3486-1411</td></tr>
<tr><td colspan="2">価　格：19,800 円(税別 1991 年 4 月 1 日現在)</td></tr>
<tr><td rowspan="2">動作環境</td><td>OS：MS-DOS Ver.2.1 以上</td></tr>
<tr><td>メモリ：640KB 以上
ハードディスク、またはフロッピーディスクドライブ 2 台</td></tr>
<tr><td rowspan="2">構成</td><td>マニュアル：2 冊 (USER'S GUIDE、REFERENCE GUIDE)</td></tr>
<tr><td>フロッピーディスク：3 枚(2HD)/4 枚(2DD)</td></tr>
<tr><td>インストール方法</td><td>1 枚目のフロッピーディスクに入っている「install.exe」を実行してください。詳しくはマニュアル(USER'S GUIDE)を参照してください。</td></tr>
<tr><td>コンパイル手順</td><td>TURBO C では、エディタなどを含めた統合環境が提供されています。統合環境では、エディタの画面からメニュー選択によってコンパイル作業を実行したり、デバック作業を行うことができます。以下にこの環境で、コンパイルする方法を示します。</td></tr>
</table>

```
  File  Edit  Run  Compile  Project  Options  Debug  Break/watch
                                       Edit
      Line 16    Col 2   Insert Indent Tab Fill Unindent    B:ADDCLOCK.C
#include <stdio.h>

main()
{
    int hour,minute;

    hour = 9 + 1;
    minute = 45 + 25;

    if (minute >= 60) {
        hour = hour+1;
        minute = minute-60;
    }

    printf("%d:%d¥n",hour,minute);
}
                              Message

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

メインメニューの「F:ファイル」から「L:詠込み」を選択し、ソースファイルを詠み込む。

メインメニューの「C:コンパイル」から「M:メイク」を選択すると、コンパイル、リンクを実行する

```
 File  Edit  Run  Compile  Project  Options  Debug  Break/watch
                                     Edit
      Line 16    Col 2   Insert Indent Tab Fill Unindent   B:ADDCLOCK.C
#include <stdio.h>

main()
{
    int hour,minut ┌───────── Compiling ─────────┐
                   │                             │
    hour = 9 + 1;  │ Main file: ¥ADDCLOCK.C      │
    minute = 45 +  │ Compiling: C:¥TC¥INCLUDE¥STDIO.H
                   │                             │
    if (minute >=  │              Total    File  │
        hour = hou │   Lines compiled: 192      192  │
        minute = m │        Warnings: 0        0    │
    }              │          Errors: 0        0    │
                   │                             │
    printf("%d:%d¥ │  Available memory: 148K     │
}                  │         Ctrl-C to quit      │
                   └─────────────────────────────┘
──────────────────────── Message ────────────────────────

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

コンパイル中の画面

```
B:¥>tc addclock.c
11:10
```

メインメニューの「R:実行」から「R:実行」を選択して、addclock.exe が実行される。

2 関数リファレンスの使い方

プログラムを組むときには、まず使いたい関数があるかどうかをリファレンスマニュアルの「機能（概要）」の項目を使って調べます。使えそうな関数が見つかったならば、その関数の引数や戻り値を確認します。あとはその呼び出し記法に合わせてプログラムを作成します。

■ Turbo C 2.0 のリファレンスマニュアルの場合	
puts	
機能	stdout に文字列を出力します。
形式	int puts(const char ＊s);
プロトタイプ	stdio.h
解説	puts は、ヌル文字 で終わる文字列 s を、標準出力ストリーム stdout に出力し、最後に改行文字をつけます。
戻り値	puts は、成功した場合は負でない値を返し、エラーの場合は EOF を返します。
可搬性	puts は UNIX システムで使用でき、ANSI-C と互換性があります。
関連項目	cputs, fputs, gets, printf, putchar

機能
関数の機能が簡単に紹介されています。まずはこの項目を見て関数を探すのがよいでしょう。
形式
この関数の使い方が書かれています。引数や戻り値の型はこの項目を参照します。
プロトタイプ
この関数のプロトタイプ宣言が書かれているヘッダファイルの名前です。この関数を使うときには、このファイルをインクルードしておきます。
解説
引数の意味や、関数の動作についての具体的な説明です。関数の使い方がわからない場合はこの項目を参照します。
戻り値
この関数が返す値についての説明です。戻り値の意味を知りたいときにはこの項目を参照します。
可搬性
他の処理系でも、この関数を利用できるかどうかを示しています。主にプログラムを移植するときなどに参考にします。
関連項目
この関数と関係が深い項目が書かれています。目的の関数が見つからない場合、それらの項目を目安に探すとよいでしょう。
例（この項目がない場合もある）
この関数を使ったサンプルが書かれています。解説や形式だけではわからないときに、参考にするとよいでしょう。

なお、Quick C の場合は、Quick Reference という簡単なマニュアルしか付いてきません。もしそれでは足りない場合は操作環境から呼び出すオンラインマニュアル等を参照してください。

また、本書で取り上げていない項目に関しては、Appendix 8 の参考文献を参照してください。

printf()関数の書式文字列

代表的なライブラリ関数である printf()の使い方について解説します。

①	②	③	④	⑤	⑥
-左詰め＋右詰め	0 を指定すると数値の入らない桁が 0 で埋まる	データを表示する最大の桁数	小数点または表示する桁数の区切り	小数点以下の桁数(数値)、表示する桁数(文字列)	データ=表現の型 d=整数、s=文字列、ld=long型または unsigned int 型等

例

printf("%d",28)	28	char s [10] ="MOJIRETSU" としてある場合		printf("%ld",0x20000)	131072
printf("%4d",28)	␣␣28	printf("%s",s)	MOJIRETSU		
printf("%04d",28)	0028	printf("%15s",s)	␣␣␣␣␣␣MOJIRETSU		
printf("%-4d",28)	28␣␣	printf("%-15s",s)	MOJIRETSU␣␣␣␣␣␣		

3 演算子書式一覧

●単項演算子

インクリメント／デクリメント演算子

演算子	意味	使用例
++ --	インクリメント デクリメント	a++または++a　(a=a+1;と同じ) a--または--a　(a=a-1;と同じ)

注意：a++と++a の違い
x = a++ …… x に a を代入してから a をインクリメント
例)a=5 のとき x=5,a=6 となる
x = ++a …… a をインクリメントしてから x に a を代入
例)a=5 のとき x=6,a=6 となる

その他の単項演算子

演算子	意味	使用例
& *	アドレス演算子 間接演算子	& a　(変数 a のアドレス) * a　(ポインタ型変数 a の指す変数の値)
+ -	+演算子 -演算子	+ a　(a の値 …… -演算子との対称性のためにある) - a　(a の符号を反転した値 …… 符号の反転)

~	1の補数	~a　(aのビットごとの反転)
!	論理否定(NOT)	!a　(aが真(0以外)ならば偽(0),偽(0)ならば真(1))
sizeof	メモリ上のサイズ	sizeof(型名)　(型名の変数のサイズ(バイト数))

●キャスト演算子

演算子	意味	使用例
(型)	型変換	i = (int)(22/7);　22/7の整数部をiに代入 f = (double)i;　iを倍精度に変換してfに代入

●算術演算子

演算子	意味	使用例
* / % + -	乗算 除算 剰余算 加算 減算	a = b * c;　bとcを掛けた値をaに代入 a = b / c;　bをcで割った値をaに代入 a = b % c;　bをcで割った余りをaに代入 a = b + c;　bとcを加えた値をaに代入 a = b - c;　bからcを引いた値をaに代入

●シフト演算子

演算子	意味	使用例
<< >>	左シフト 右シフト	a = b << 2;　(bを2ビット左シフトしてaに代入) a = b >> 3;　(bを3ビット右シフトしてaに代入)

●関係演算子

演算子	意味	使用例
< > <= >=	より小さい より大きい 小さいか等しい 大きいか等しい	a < b　(aがbより小さい場合は真(1)、そうでない場合は偽(0)) a > b　(aがbより大きい場合は真(1)、そうでない場合は偽(0)) a <= b　(aがbより小さいか等しい場合は真(1)、そうでない場合は偽(0)) a >= b　(aがbより大きいか等しい場合は真(1)、そうでない場合は偽(0))

●等値演算子

演算子	意味	使用例
== !=	等しい 等しくない	a == b　(aがbと等しい場合は真(1)、そうでない場合は偽(0)) a != b　(aがbと等しくない場合は真(1)、そうでない場合は偽(0))

●ビット演算子

演算子	意味	使用例
& ^ |	ビットAND ビットEXOR ビットOR	a & b　(aとbのビットごとの論理積) a ^ b　(aとbのビットごとの排他的論理和) a | b　(aとbのビットごとの論理和)

●論理演算子

<table>
<tr><th>演算子</th><th>意味</th><th>使用例</th></tr>
<tr><td>&&</td><td>論理積(AND)</td><td>a && b　(a と b がともに真(0 以外)ならば真(1)、そうでないならば偽(0))</td></tr>
<tr><td>||</td><td>論理和(OR)</td><td>a || b　(a と b のどちらかが真(0 以外)ならば真(1),そうでないならば偽(0))</td></tr>
</table>

●代入演算子

<table>
<tr><th>演算子</th><th>意味</th><th>使用例</th></tr>
<tr><td>=</td><td>右辺を左辺に代入</td><td>a = b;</td></tr>
<tr><td>*=</td><td>左辺と右辺を乗算し、左辺に代入</td><td>a *= b;　(a = a * b;と同じ)</td></tr>
<tr><td>/=</td><td>左辺を右辺で除算し、左辺に代入</td><td>a /= b;　(a = a / b;と同じ)</td></tr>
<tr><td>%=</td><td>左辺を右辺で剰余し、左辺に代入</td><td>a %= b;　(a = a % b;と同じ)</td></tr>
<tr><td>+=</td><td>左辺と右辺を加算し、左辺に代入</td><td>a += b;　(a = a + b;と同じ)</td></tr>
<tr><td>-=</td><td>左辺を右辺で減算し、左辺に代入</td><td>a -= b;　(a = a - b;と同じ)</td></tr>
<tr><td><<=</td><td>左辺を右辺で左シフトし、左辺に代入</td><td>a <<= b;　(a = a <<b;と同じ)</td></tr>
<tr><td>>>=</td><td>左辺を右辺で右シフトし、左辺に代入</td><td>a >>= b;　(a = a >>b;と同じ)</td></tr>
<tr><td>&=</td><td>左辺と右辺を AND し、左辺に代入</td><td>a &= b;　(a = a & b;と同じ)</td></tr>
<tr><td>^=</td><td>左辺と右辺を EXOR し、左辺に代入</td><td>a ^= b;　(a = a ^ b;と同じ)</td></tr>
<tr><td>|=</td><td>左辺と右辺を OR し、左辺に代入</td><td>a |= b;　(a = a | b;と同じ)</td></tr>
</table>

●条件演算子（三項演算子）

演算子	意味	使用例	
? :	条件式を作る	式 1?式 2:式 3	(式 1 の値が真(0 以外)ならば式 2 の値、偽(0)ならば式 3 の値を返す)
		x =(a > b)?a:b;	(a が b よりも大きければ x には a の値が、そうでなければ x には b の値が代入される)

注意: '(a > b)' のカッコはなくても構わない。

●カンマ演算子

演算子	意味	使用例	
,	複数の式の実行	(式 1,式 2)	(式 1、式 2 の順に式を評価する)
		x =(a = 2,a +3);	(a に 2 を代入、a に 3 を加算した値を x に代入)

注意: '(a = 2,a +3)'のカッコは必要。

4 演算子優先順位一覧

<table>
<tr><th>優先順位</th><th>演算子の種類</th><th>演算子</th><th>結合規則</th></tr>
<tr><td>高い</td><td>式</td><td>() [] -> .</td><td>→</td></tr>
<tr><td>↑</td><td>単項演算子</td><td>! ~ ++ -- + - * & (型名) sizeof</td><td>←</td></tr>
<tr><td></td><td>乗除算</td><td>* / %</td><td>→</td></tr>
<tr><td></td><td>加減算</td><td>+ -</td><td>→</td></tr>
<tr><td></td><td>シフト演算子</td><td><< >></td><td>→</td></tr>
<tr><td></td><td>関係演算子</td><td>< <= > >=</td><td>→</td></tr>
<tr><td></td><td>等値演算子</td><td>== !=</td><td>→</td></tr>
<tr><td></td><td>ビット AND</td><td>&</td><td>→</td></tr>
<tr><td></td><td>ビット EXOR</td><td>^</td><td>→</td></tr>
<tr><td></td><td>ビット OR</td><td>|</td><td>→</td></tr>
<tr><td></td><td>論理 AND</td><td>&&</td><td>→</td></tr>
<tr><td></td><td>論理 OR</td><td>||</td><td>→</td></tr>
<tr><td></td><td>条件演算子</td><td>?:</td><td>←</td></tr>
<tr><td>↓</td><td>代入演算子</td><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>←</td></tr>
<tr><td>低い</td><td>カンマ演算子</td><td>,</td><td>→</td></tr>
</table>

＊結合規則とは、同順位(同じ列)の演算子が並んだ場合、左から右(→)への順番で演算する、右から左(←)の順番で演算するかを表す。

5 文法要覧

●文とブロック（本文は43、45ページを参照してください）

文
文とは関数呼出や代入等の命令の1つの要素であり「;」で終わります。
ブロック
ブロックとは文の集まりで「{」と「}」でくくられています。
例

```
int test()
{
   int a;                この部分がブロック
   a = getchar();        （ブロックの中のそれぞれが文）
   putchar(a);
   return(a);
}
```

＊以下の書式で文とあるところはいずれもブロックを書くことができます。

●制御構造

if、else、else if（本文は122ページを参照してください）

書式　if (式) 文
説明　式の値が真(0以外)ならば文1を実行する。
例

```
if (a > m) {
        m = a;
}
```

書式　if (式) 文 1 else 文 2
説明　式の値が真(0 以外)ならば文 1 を、偽(0)ならは文 2 を実行する。
例
```
if (a > b) {
        x = a - b;
} else {
        x = b - a;
}
```

書式　if (式 1) 文 1 else if (式 2) 文 2
説明　式 1 の値が真(0 以外)ならば文 1 を、偽(0)でかつ式 2 の値が真(0 以外)ならば文 2 を実行する。
例
```
if (a == b) {
        x = 0;
} else if (a > b) {
        x = 1;
} else {
        x = -1;
}
```

switch（本文は 125 ページを参照してください）

書式
```
switch (式) {
case 定数式: 文の並び
case 定数式: 文の並び
default: 文の並び
}
```
説明　式の値が定数式と一致した文を実行する。一致しない場合は default:の次に書かれた文を実行する。
例
```
switch (x) {
case 0:
        puts("a == b");
        break;…………………………switch から抜けるための break 文
case 1:
        puts("a > b");
        break;
case -1:
        puts("a < b");
        break;
default:
        puts("error!");
        break;
}
```

while、for、do〜while（本文は 111 ページを参照してください）

書式　while (式) 文
説明　式の値が真(0 以外)ならば文を実行する。これを式の値が偽(0)になるまで繰り返す。
例
```
while (*p == ' ') {
        p++;
}
```

書式　for (式 1;式 2;式 3) 文
説明　まず式 1 を計算する。つぎに式 2 の値が真(0 以外)ならば文を実行する。最後に式 3 を計算する。これを式 2 の値が偽(0)になるまで繰り返す。
例
```
for (n = 0; s[n] != '\0'; n++) {
}
```

書式　do 文 while (式);
説明　文を実行してから式を評価する。式の値が偽(0)になるまで繰り返す。
例
```
do {
        *p++ = *s;
} while (*s++ != '\0');
```

break（本文は 119 ページを参照してください）、 continue

書式　break;

説明　最も内側のループ(while,for,do～while)または switch から抜ける。

例
```
for (i = strlen(s) - 1; i > = 0; i--)  {
        if (!isspace(s[i] )  {
                braek;
        }
        s[i] = '\0';
}
```

書式　continue;

説明　最も内側のループ(while,for,do)の最後に制御を移す。

例
```
for (i = 0; s[i]  != '\0'; i++)  {
        if (!iskanji(s[i] ))  {
                continue;
        }
        i++;
}
```

return（本文は 87 ページを参照してください）

書式　return 式;

説明　式の値を戻り値として呼び出した関数に返す。

例
```
int test(int a)
{
        int i;
        i=a * a;
        return(i); ←( )はなくてもよい
}
```

goto, ラベル文

書式　goto ラベル;

説明　無条件にラベルのある行に実行を移す。

例
```
for (i = 0; i  < 100; i++)
        for (j = 0; j  < 100; j++)
                if (a [i]  == b [j] )
                        goto next; ← break だと 1 つめの for を抜けられない
next:←ラベル
```

書式　ラベル: 文

説明　goto の飛び先に用いる。switch 文の case や default もラベルの一種である。

●データ型

	整数型(MS-C 6.0の場合)		整数型(SUN OS の場合)	
データ型	サイズ	値の範囲	サイズ	値の範囲
char	1 バイト	-127 ～ 127	1 バイト	-127 ～ 127
short int	2 バイト	-32767 ～ 32767	2 バイト	-32767 ～ 32767
int	2 バイト	-32767 ～ 32767	4 バイト	-2147483647 ～ 2147483647
long int	4 バイト	-2147483647 ～ 2147483647	4 バイト	-2147483647 ～ 2147483647
unsigned char	1 バイト	0 ～ 255	1 バイト	0 ～ 255
unsigned short int	2 バイト	0 ～ 65535	2 バイト	0 ～ 65535
unsigned int	2 バイト	0 ～ 65535	4 バイト	0 ～ 4294967295
unsigned long int	4 バイト	0 ～ 4294967295	4 バイト	0 ～ 4294967295

浮動小数点型（MS-C 6.0 の場合）				
データ型	サイズ	有効数字	最小値	最大値
float	4 バイト	6 桁	約 1.18E-38	約 3.40E+38
double	8 バイト	15 桁	約 2.23E-308	約 1.80E+308
long double	10 バイト	19 桁	約 3.36E-4932	約 1.19E+4932

●変数(配列)の宣言方法

変数の宣言
　型名 変数名 [= 初期値]； または，型名 変数名，変数名，……;
配列変数
　型名 変数名[要素数] [= {初期値, 初期値, ...}] ;([要素数]の[ ]は省略できない)
ポインタ変数
　型名 *変数名 [= 初期値]；
構造体変数
　struct {型名 メンバ;...} 変数名;
　struct タグ名 {型名 メンバ;...} 変数名;
　struct タグ名 {型名 メンバ;...} ;
　struct タグ名 変数名;

● typedef の宣言方法

新しい型の定義方法
　typedef 変数宣言;(変数名の部分に新しい型名を書く)
新しい型の変数の宣言方法
　新しい型名 変数名;

●関数の使い方

定義方法
　型名 関数名([型名 変数 [, 型名 変数 ...]]) { 関数本体 }
呼びだし方法
　関数名(式 [, 式 ...]);
戻り値を返す
　return 式;
プロトタイプ宣言
　型名 関数名([型名 [引数] [,型名 [引数] ...]]);

6 キーワード一覧

次に示す識別子はキーワード（予約語）として定められているので、別の用途に使うことはできません。これらは、C言語のデータ型、制御構造、演算子として使われます。

●データ型に関するもの

void、char、int、float、double、short、long、unsigned、signed、auto、static、register
const、volatile、extern、typedef、struct、union、enum

●制御構造に関するもの

if、else、switch、case、default、for、while、do、goto、break、continue、return

●演算子に関するもの

sizeof

7 文字キャラクタセット（ASCII コード）

下位＼上位	0	1	2	3	4	5	6	7
0	(0)	(16)	空白 (32)	0 (48)	@ (64)	P (80)	' (96)	p (112)
1	(1)	(17)	! (33)	① (49)	Ⓐ (65)	Q (81)	ⓐ (97)	q (113)
2	(2)	(18)	" (34)	2 (50)	B (66)	R (82)	b (98)	r (114)
3	(3)	(19)	# (35)	3 (51)	C (67)	S (83)	c (99)	s (115)
4	(4)	(20)	$ (36)	4 (52)	D (68)	T (84)	d (100)	t (116)
5	(5)	(21)	% (37)	5 (53)	E (69)	U (85)	e (101)	u (117)
6	(6)	(22)	& (38)	6 (54)	F (70)	V (86)	f (102)	v (118)
7	a (7)	(23)	' (39)	7 (55)	G (71)	W (87)	g (103)	w (119)
8	b (8)	(24)	((40)	8 (56)	H (72)	X (88)	h (104)	x (120)
9	t (9)	(25)	) (41)	⑨ (57)	I (73)	Y (89)	i (105)	y (121)
A	n (10)	(26)	* (42)	: (58)	J (74)	Ⓩ (90)	j (106)	z (122)
B	(11)	(27)	+ (43)	; (59)	K (75)	[(91)	k (107)	{ (123)
C	f (12)	(28)	, (44)	< (60)	L (76)	(92)	l (108)	| (124)
D	r (13)	(29)	- (45)	= (61)	M (77)	] (93)	m (109)	} (125)
E	(14)	(30)	. (46)	> (62)	N (78)	^ (94)	n (110)	~ (126)
F	(15)	(31)	/ (47)	? (63)	O (79)	_ (95)	o (111)	(127)

文字コードは 16 進数で表すことが多いので 16 進表記の表にしてあります。
() 内には、10 進数で表した文字コードを示してありますので参照してください。

① アルファベットは A から Z まで順番に並んでいるので、

```
if(c >= 'A' && c  <= 'Z')…
```

という式でアルファベットの大文字かどうかを判断できる。同様に、

```
if(c >='0' && c  <= '9')…
```

という式で数字であるかどうかを判断できる。

② 大文字と小文字はちょうど 32 ずつ離れているので、大文字に 32 を加えると小文字に変換できる。逆に、小文字から 32 を引くと大文字に変換できる。

```
c+= 32; /*小文字への変換*/
c -= 32; /*大文字への変換*/
```

③ 数字は '0' から順番に並んでいるので、次のような式で数値に変換できる。
c を数字とすると、

```
i = c - '0';
```

（本文 176 ページを参照）

- 文字「'」は「'\''」と書く
- 文字列中で「"」を使うには「"…\"…"」とする
- 文字「\」は「'\\'」と書く
- 文字列中で文字「\」を使うには「"…\\…"」とする
- 「{」という記号を扱えないコンピュータのために「??<」という記法で「{」を表すことになっている。このため「"??"」のように「?」を2つ連続で使う文字列では「"\?\?"」とする。
- 「リターンキー」に対応する特殊文字は「'\n'」と書く。「エスケープキー」に対応する特殊文字は文字コードを使って「'\033'」(8進数表記)や「'\x1b'」(16進数表記)のように書く。

●特殊文字

プログラム中で特殊文字を使用する場合、次のように表す。

特殊文字	表現方法
'	\'
"	\"
?	\?
\	\\
ベル	\a
後退(バックスペース)	\b
改頁(フォームフィード)	\f
改行	\n
復帰	\r
水平タブ	\t
垂直タブ	\v
空白	␣
拡張制御(エスケープ)	\033
8進数	\000
16進数	\xHH

8　参考文献一覧

●C言語をさらに学習するには

書名	内容
『実習C言語　改訂新版』 三田典玄著/アスキー出版局刊	本書で解説しなかった部分も含め、C言語の文法のすべてを解説している。本格的にC言語のプログラム作成をはじめるには、このレベルの本を読んでおくことが望ましい。
『プログラミング言語C　第2版』 B.W.Kernighan・D.M.Ritchie共著/共立出版刊	C言語の開発者自ら、プログラムに対する考え方とともに解説している。プログラミングの経験をある程度積んでから読むとよい。
『応用C言語　改訂新版』 三田典玄著/アスキー出版局刊	プログラムの開発環境、日本語処理、MS-DOSやUNIXの機能とその利用方法など多くの側面から実践的にC言語を使ったプログラミングテクニックを解説している。
『Cプログラミングの落とし穴』 A.Koenig著/トッパン刊	C言語を使ったプログラム作成時にミスを起こしやすいところを詳しく解説している。C言語の学習につまずいてしまった時に頼りになる本。

●データ構造とアルゴリズムを学習するには

書名	解説
『C-データ構造とプログラム』 Leendert Ammeraal 著/オーム社刊	基本的なデータ構造とアルゴリズムを豊富な例題とともに解説した好著。専門用語が多くとっつきにくいかもしれないが、情報をプログラムで表現する考え方が身につく。
『思考ゲームプログラミング』 森田和郎・国枝交子・津田伸秀共著/アスキー出版局刊	オセロゲームの作成を通じて実践的なアルゴリズムとプログラミング方法を解説している。
『ソフトウェア作法』 Brian W. Kernighan・P.J. Plauger 共著/共立出版刊	プログラム作成環境に有用なソフトウェアツールを作成する技法を解説した名著。プログラムは ratfor という言語で書かれているが C 言語によく似ているので理解しやすいだろう。

●マシン語を学習するには

書名	解説
『はじめて読むマシン語』 村瀬康治著/アスキー出版局刊	8 ビットの Z80CPU のマシン語の解説を通じてコンピュータの仕組みを解説する名著。
『はじめて読む 8086』 村瀬康治監修・蒲地輝尚著/アスキー出版局刊	『はじめて読むマシン語』の 16 ビット版。8086、80286、80386 などの CPU を使った MS-DOS マシンの仕組みを学習するのに最適の書。
『はじめて読む MASM』 蒲地輝尚著/アスキー出版局刊	『はじめて読む 8086』の続編で、アセンブラを使ったプログラミングや C 言語とマシン語のリンク方法などを解説している。

●オペレーティングシステムについて学習するには

書名	解説
『応用 MS-DOS　改訂新版』 村瀬康治著/アスキー出版局刊	MS-DOS の仕組みや内部構造をわかりやすく解説している。MS-DOS の機能やその利用方法も詳しく解説しているので、MS-DOS プログラミングの入門書として最適。
『MS-DOS プログラミングテクニック』 アスキー書籍編集部編/アスキー出版局刊	MS-DOS の機能とプログラムからの利用方法を実践的なテクニックとともに解説している。日本語の処理方法も詳しい。
『UNIX システムコールプログラミング』 Marc J. Rochkind 著/アスキー出版局刊	UNIX オペレーティングシステムの機能と、プログラミングのノウハウを解説している。

● C++言語を学習するには

書名	解説
『プログラミング言語 C++』 Bjarne Stroustrup 著/トッパン刊	C++言語の開発者による解説書。C 言語で十分経験を積んでから挑戦しよう。

Index

L

M

N

P

R

S

T

U

V

W

Y

記号

ア

カ

サ

タ

ナ

ハ

執筆者紹介

蒲地 輝尚(かまち てるひさ)

1964年長崎市生まれ。鹿児島市で育つ。

東京大学工学部卒。現在、大手電気メーカー勤務。

著書に「はじめて読む8086」、「はじめて読むMASM」、執筆協力に「応用MS-DOS改訂新版」(いずれもアスキー出版局発行)がある。

MS-DOSマシン、Macintosh、UNIXワークステーションなど多種のコンピュータを、それぞれの得意な面を活かしながら利用している。

世界中のコンピュータをネットワークに接続して新しい生活環境を作るための技術や、身近な機器をうまく連携させて利用する環境について研究中。

技術／編集協力

西　克弘/青柳貴洋/富田憲範

プログラミング環境のスタンダード
はじめて読むC言語

1991年5月11日　初版発行
1992年7月21日　第1版第5刷発行
定価1,800円（本体1,748円）

著　者　蒲地 輝尚
発行者　藤井 章生
編集人　井芹 昌信
発行所　**株式会社アスキー**
〒107-24　東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
大代表　(03)3486-7111
出版営業部　(03)3486-1977（ダイヤルイン）
第一書籍編集部　(03)3797-3225（ダイヤルイン）

制　作　株式会社　アート・コンストラクション・センター
印　刷　株式会社　加藤文明社印刷所

編　集　佐藤 英一／斉藤 邦男

ISBN4-7561-0071-6　　Printed in Japan

●11600